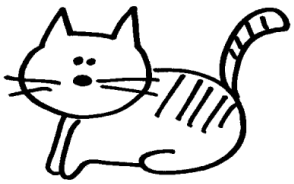
Un Compilatore ad Oggetti per Kitten

Fausto Spoto
Dipartimento di Informatica
Università di Verona



Presentazione

Scrivere un nuovo libro per un corso di compilazione sembra un'operazione persa in partenza. È da venti anni che il *dragon book* di Aho, Sethi e Ullman [1] rimane sul mercato come l'unico e insostituibile riferimento per chi si cimenta nella rara e tuttora difficile arte di scrivere un compilatore. Ma gli anni passano ed ecco che da una parte i corsi di laurea relegano la compilazione nello spazio di poche lezioni; dall'altra l'evoluzione tecnologica dei linguaggi di programmazione rende il *dragon book* carente quanto al trattamente della compilazione dei linguaggi *ad* oggetti e della stessa compilazione *con* gli oggetti. Perché la diffusione dei linguaggi a oggetti non è stata un cambiamento di facciata per quanto riguarda il mondo dei compilatori: essa valorizza il sistema di tipaggio del linguaggio; richiede di spostare a tempo di esecuzione controlli e legami altrimenti effettuati a tempo di compilazione; rende desiderabili nuove tecniche di analisi e ottimizzazione del codice; infine, la stessa implementazione ad oggetti di un compilatore rende più credibile la sua correttezza e permette l'implementazione di ottimizzazioni tramite specializzazioni di classi e metodi della sintassi astratta.

Ecco quindi nascere la necessità di questo libro sulla compilazione ad oggetti di un linguaggio ad oggetti. Non mancano certo altri tentativi in questa direzione. Fra i tanti, non possiamo dimentivare il libro di Appel [2], che ha inizialmente ispirato questo testo. Ma le nostre soluzioni per il tipaggio statico e la generazione del codice non sono assolutamente assimilabili alle tecniche, scarsamente ad oggetti, dell'Appel.

Questo libro si offre come supporto allo studente impegnato in un corso di compilazione che a Verona è organizzato in appena una quarantina di ore di lezione, laboratorio incluso. Le scelte che hanno guidato la selezione degli argomenti trattati sono quelle dell'utilizzo intensivo della programmazione ad oggetti; dell'evidenziazione continua della relazione biunivoca fra teoria e implementazione di un compilatore; della maggiore importanza data alle fasi avanzate della compilazione, come controllo dei tipi, generazione e ottimizzazione del codice, rispetto alle prime fasi di analisi lessicale e sintattica.

Un ringraziamento va agli studenti che hanno seguito a Verona il mio corso di compilazione negli ultimi anni. È dall'interazione che ho avuto con loro che deriva la presentazione degli argomenti trattati in questo libro. Sono loro e i loro dubbi che mi hanno spinto a scrivere un libro e del codice che fosse facilmente comprensibile e meno ambiguo possibile. La scarsa presenza di bug nel compilatore Kitten è il risultato della loro, spesso involontaria, verifica.

Fausto Spoto
Verona, gennaio 2007

Indice

Capitolo 1

Introduzione a Kitten

Kitten è il linguaggio di programmazione di cui in questo libro vogliamo descrivere un compilatore scritto in Java. Benché quindi questo libro non sia centrato solo su Kitten, è necessario cominciare a prendere familiarità con tale linguaggio, in modo da essere coscienti di quello che è lo scopo del nostro compilatore. Il fine di questo capitolo è di descrivere l'installazione del compilatore Kitten e il linguaggio Kitten stesso. Il compilatore ci permetterà di compilare ed eseguire tutti i programmi Kitten di esempio che incontreremo in queste pagine.

Kitten è un semplice linguaggio di programmazione imperativo a oggetti. È un linguaggio *imperativo* poiché l'esecuzione dei programmi Kitten consiste in una sequenza di passi specificati da *comandi* e ciascun comando determina una modifica dello stato dell'esecutore. È un linguaggio *a oggetti* poiché lo stato dell'esecutore lega le variabili del programma a degli *oggetti* appunto, cioè zone di memoria contenenti informazioni e che reagiscono all'invocazione di *metodi*. Va detto che Kitten non è un linguaggio a oggetti *puro*, nel senso che alcune variabili possono non essere legate a oggetti, ma piuttosto a *valori primitivi*. Esempi di valori primitivi sono gli interi e i numeri in virgola mobile. Esistono pochissimi linguaggi di programmazione puramente ad oggetti. In particolare, va ricordato Smalltalk [3]. Java [4] non è puramente ad oggetti, perché anch'esso ha dei tipi primitivi (che però coesistono con delle versioni ad oggetti dei tipi primitivi, cioè le *classi involucro* tipo `java.lang.Integer` e simili). Il motivo per cui i linguaggi ad oggetti tendono a non essere puri è che i valori primitivi sono gestibili molto più efficientemente che gli oggetti.

Possiamo immaginare Kitten come una versione semplificata di Java, sia da un punto di vista sintattico che semantico. Lo scopo di Kitten è infatti quello di essere un linguaggio abbastanza semplice da potere essere compilato ed eseguito senza eccessive complicazioni, ma al contempo sufficientemente rappresentativo dei problemi che si presentano nella compilazione dei linguaggi

di programmazione attuali tipo Java. In Kitten mancano all'appello aspetti *secondari* di Java, come *i modificatori di visibilità* (`public`, `protected`, `private`), i metodi e i campi `static`, le classi astratte e le interfacce, i package, i campi costanti, le eccezioni e i finalizzatori.

Lo studente potrebbe domandarsi perché non si è scelto Java come linguaggio da compilare, al posto di Kitten. In fin dei conti, Kitten non è utilizzato se non in questo corso e quindi la sua importanza decade con il corso stesso. Il motivo è che la compilazione di Java è estremamente complessa per essere descritta in un breve corso di compilazione essendo Java, come abbiamo visto, molto più complicato di Kitten. È invece più interessante chiedersi perché non si sia scritto un compilatore Kitten *in Kitten*. Questo ci avrebbe permesso, per esempio, di compilare il nostro compilatore con se stesso! Il motivo questa volta è che Kitten è troppo povero per permettere la definizione di un compilatore senza eccessivi sforzi di programmazione. Per esempio, l'assenza dei modificatori di visibilità e del *constructor chaining* priva la gerarchia delle classi di ogni potere di incapsulazione, aprendo la strada a codice criptico e scarsamente controllabile. L'assenza dei package impedisce ogni strutturazione del progetto. Va inoltre ricordato che esistono e che utilizzeremo degli strumenti di sviluppo di compilatori scritti in Java, come JLex (Capitolo 2) e JavaCup (Capitolo 3), e che tali strumenti andrebbero riscritti ex-novo in Kitten.

1.1 Il compilatore Kitten

Il compilatore Kitten è scritto in Java. Si tratta, allo stato attuale, di uno strumento che permette di compilare dei sorgenti Kitten trasformandoli in file `.class` eseguibili da una qualsiasi Java virtual machine. Conseguentemente, sia il compilatore Kitten che il risultato della compilazione dei sorgenti Kitten possono essere eseguiti su qualsiasi architettura e sistema operativo, purché sia installato un compilatore e interprete Java. Le istruzioni che seguono sono pensate per un'architettura basata su Unix, tipo Linux. La compilazione sotto Windows del compilatore Kitten è possibile ma è non ancora stata automatizzata.

Supponiamo di avere scaricato il file di installazione del compilatore Kitten. Esso sarà un file dal nome `kitten_XXX.tgz`, dove `XXX` è la versione del compilatore Kitten che si sta installando. Tale file dovrà essere posizionato nel punto in cui si vuole installare il compilatore. Basta quindi dare i seguenti comandi per installarlo:

```
tar -zxf kitten_XXX.tgz
cd kitten_XXX
make generatejb
```

Con il primo comando scompattiamo il file di installazione. Con il secondo entriamo nella directory così creata. Con il terzo compiliamo il compilatore. Se non è stato segnalato alcun errore, il compilatore Kitten è a questo punto utilizzabile.

```
class Miao {
  method void main() {
    "miao\n".output()
  }
}
```

Figura 1.1: Un programma Kitten che stampa una stringa e termina.

1.2 Il nostro primo programma Kitten

La Figura 1.1 mostra un esempio di programma Kitten. Si tratta di un programma che, una volta compilato ed eseguito, stampa a video la stringa `miao` seguita da un ritorno carrello. Supponiamo di avere inserito il programma in Figura 1.1 dentro a un file testo di nome `Miao.kit` che si trova nella sottodirectory `testcases` della directory `kitten_XXX`. La sottodirectory `testcases` è quella che utilizzeremo per i nostri esperimenti, in modo da non sporcare la directory principale di Kitten. Essa non ha comunque nulla di speciale: potevamo scegliere un qualsiasi altro nome. È invece importante che il nome del file `Miao.kit` termini con `.kit`. In caso contrario esso non verrà riconosciuto dal compilatore come un file testo che contiene un sorgente Kitten. Si entri quindi nella sottodirectory `testcases`. La compilazione del nostro primo programma Kitten si ottiene scrivendo

```
java -cp .. Kitten Miao.kit
```

Questo comando chiama l'interprete Java chiedendo di eseguire un programma di nome `Kitten`. Si tratta del compilatore Kitten che abbiamo appena compilato. Dal momento che esso si trova dentro la directory `kitten_XXX`, siamo costretti a indicare che i file Java devono essere cercati in tale directory, cioè nella directory padre di `testcases`, in cui siamo. Questo si ottiene con lo switch `-cp ..` che specifica il *classpath* per l'interprete Java. Infine passiamo un parametro al compilatore Kitten. Si tratta del nome del file che contiene il programma Kitten che vogliamo compilare. In questo caso, si tratta di `Miao.kit`, il file sorgente Kitten da compilare.

Il precedente comando è così frequente che è stata prevista una macro per semplicarne l'esecuzione. Basta infatti digitare

```
./compile Miao.kit
```

per ottenere esattamente lo stesso effetto. Si noti che, per ulteriore semplificazione, questa macro deve essere invocata dalla directory `kitten_XXX` e non da `testcases`.

In ogni caso, se tutto è andato a posto, il compilatore Kitten dovrebbe aver compilato il file in Figura 1.1 comunicando a video delle informazioni del tipo:

```
Parsing and type-checking completed             [419ms]
Translation into Kitten bytecode completed      [66ms]
Kitten bytecode dumping in dot format completed [15ms]
Java bytecode generation completed              [478ms]
Total compilation time was 964ms
```

La prima riga ci informa che Kitten ha effettuato un'analisi sintattica (*parsing*) sul file `Miao.kit`, al fine di garantire che non contenga errori di sintassi. A questa fase ne è seguita una di verifica semantica (*type-checking*). Il tutto ha richiesta 419 millisecondi. A queste due fasi ne è seguita una in cui il nostro programma è stato tradotto in un linguaggio chiamato *Kitten bytecode*. Si tratta di un linguaggio molto simile al Java bytecode, sul quale è possibile ragionare per effettuare eventuali ottimizzazioni. Tale bytecode è stato anche salvato su disco in formato `dot`, un formato di descrizione di grafi che ci permette di visionare il risultato di questa fase della compilazione. Infine, il Kitten bytecode è stato tradotto in Java bytecode, il quale può finalmente essere eseguito.

Sono tre i file che sono stati in effetti compilati in Java bytecode. Oltre a `Main.kit`, come ci aspettavamo, ci sono anche `Object.kit` e `String.kit`. Questi sono due file Kitten forniti insieme al compilatore. Il primo descrive la classe `Object`, cioè la superclasse di tutte le classi Kitten. Esso è stato compilato poiché la classe `Miao.kit` in Figura 1.1 estende (implicitamente) la classe `Object.kit`. Il secondo descrive la classe delle stringhe. Esso è stato compilato poiché il programma in Figura 1.1 chiama il metodo `output()` sulle stringhe e tale metodo è definito proprio dentro `String.kit`.

L'esecuzione del programma in Figura 1.1 può a questo punto essere effettuata eseguendo il file `Miao.class` generato dalla compilazione. Essendo quest'ultimo un file Java bytecode, esso può essere eseguito da una qualsiasi Java virtual machine. Per esempio, basta scrivere, dalla directory `testcases`,

```
java Miao
```

per vedersi stampare a video la stringa `miao`. Anche per questa operazione è stata prevista una macro, che per ulteriore semplificazione va eseguita dalla directory `kitten_XXX`. Essa è

```
./run Miao
```

Adesso che siamo riusciti a compilare ed eseguire un programma Kitten, proviamo a guardarne più da vicino il sorgente in Figura 1.1 per iniziare a comprendere in cosa Kitten rassomiglia a Java o si differenzia da esso.

La Figura 1.1 definisce una *classe* Kitten di nome `Miao`, cioè una matrice che può essere usata per generare degli *oggetti* di tale classe. Tali oggetti sono detti *istanze* della classe. Una classe può avere dei *costruttori* che specificano come generare degli oggetti di tale classe. Serve almeno un costruttore per potere creare istanze di una classe. Dal momento che nessun costruttore è presente in Figura 1.1, nessuna istanza della classe `Miao.kit` può essere creata. Una classe può anche avere dei *metodi*, cioè del codice etichettato con un nome che viene eseguito al momento dell'*invocazione* del metodo. Fin qui, troviamo solo somiglianze con Java. Ma guardando attentamente la Figura 1.1 notiamo anche molte differenze. Per esempio, i metodi sono introdotti dalle parole chiave `method`. Inoltre, il punto e virgola, che in Java è un *terminatore* di comandi, è invece un *separatore* di comandi in Kitten. Conseguentemente, non si deve mettere alcun punto e virgola alla fine del metodo `main` in Figura 1.1. Inoltre, va osservato che non ci sono costruttori di default in Kitten e che quindi una classe *deve* avere il costruttore vuoto (cioè senza parametri) per potere essere istanziata. Su tali istanze si possono poi chiamare i metodi della classe. Fa eccezione il metodo `main` che può essere chiamato senza avere a disposizione alcuna istanza della

```
class Fibonacci {
  constructor() {}

  method int fib(int n)
    if (n = 0 | n = 1) then return 1
    else return this.fib(n - 1) + this.fib(n - 2)

  method void main() {
    String s := new String();
    "Insert a relatively small number: ".output();
    s.input();
    "Fibonacci(".concat(s).concat(") = ".concat
      (new Fibonacci().fib(s.toInt()))).output();
    "\n".output()
  }
}
```

Figura 1.2: La funzione di Fibonacci in Kitten.

classe. Il metodo `main`, senza parametri e con tipo di ritorno `void`, è in effetti quello che viene eseguito quando si esegue un programma Kitten. Al momento dell'invocazione di tale metodo, non esiste ancora alcuna istanza della classe.

Si può dire, con terminologia presa in prestito da Java, che il metodo `main` di Kitten è *statico*, cioè invocato sulla classe piuttosto che sulle istanze della classe. Si noti che esso è però l'unico metodo Kitten ad avere tale caratteristica: in Kitten non esiste modo di dichiarare i metodi come *statici*. Essi saranno sempre implicitamente non *statici*.

1.3 Un esempio più complesso

Si consideri il programma in Figura 1.2. Assumiamo che esso sia scritto all'interno di un file testo di nome `Fibonacci.kit`. Il suo metodo `main` chiede all'utente di immettere un numero intero (positivo) `s` e quindi stampa l'`s`-esimo numero di Fibonacci. Si noti che `s` è una stringa, e che è possibile leggere l'input da tastiera e memorizzarlo dentro a una stringa tramite il metodo `input()` della classe `String.kit`. Va osservato che, come per tutte le invocazioni di metodo, la variabile `s` deve contenere un oggetto affinché su di essa si possa chiamare il metodo `input()`; essa non deve contenere `nil`, pena un errore a tempo di esecuzione del programma. Sulle stringhe è disponibile anche il metodo `toInt()` che trasforma la stringa nell'intero corrispondente, se possibile. Esiste anche il metodo `concat()` che restituisce la concatenazione di due stringhe o di una stringa con un intero. Tutti questi metodi sono stati usati in Figura 1.2. La Figura 1.3 descrive tutti i metodi della classe Kitten `String.kit`.

s`.length()`	restituisce la lunghezza (numero di caratteri) della stringa *s*
s`.toInt()`	restituisce l'intero rappresentato dalla stringa *s*; restituisce 0 se *s* non rappresenta un intero
s`.toFloat()`	restituisce il float rappresentato dalla stringa *s*; restituisce 0.0 se *s* non rappresenta un float
s`.equals(`*s*′`)`	restituisce *true* se e solo se le stringhe *s* ed *s*′ sono sintatticamente identiche
s`.input()`	memorizza dentro la stringa *s* una sequenza di caratteri letti da tastiera fino al primo newline (escluso)
s`.output()`	stampa a video la stringa *s*
s`.concat(X)`	restituisce la concatenazione della stringa *s* con *X*, che può essere un'altra stringa, un intero, un float o un booleano. Né *s* né *X* sono modificati

Figura 1.3: I metodi della classe Kitten `String.kit`.

La Figura 1.2 mostra come sia possibile creare un *oggetto* tramite l'espressione `new`. Esattamente come in Java, tale istruzione chiama il corrispondente costruttore della classe, in questo caso il costruttore di `Fibonacci` e quello di `String.kit` senza parametri. La stessa figura mostra come sia possibile definire un metodo ricorsivo `fib`. Si noti che, a differenza di Java, non è possibile lasciare sottinteso il riferimento `this` quando si chiama un metodo sull'oggetto corrente (o se ne modifica un campo). Si noti infine che la disgiunzione di due condizioni booleane si ottiene con la barretta `|` e che nel comando condizionale è obbligatorio usare la parola chiave `then`, che è invece sottintesa in C e Java.

1.4 Le diverse compilazioni del compilatore Kitten

Nella Sezione 1.1 abbiamo usato il comando `make generatejb` per compilare il compilatore Kitten. Tale comando esegue una sequenza di comandi di compilazione specificati dentro al file `makefile` e il cui risultato è la compilazione del compilatore Kitten in modo che esso possa poi generare il Java bytecode dei file sorgenti. Esistono altri modi per compilare il compilatore Kitten, in modo da attivare più o meno fasi della compilazione. Esse sono le seguenti:

`make lexical:` Compila il compilatore Kitten in modo che esso esegua la sola analisi lessicale del sorgente da compilare (Capitolo 2);

`make syntactical:` Compila il compilatore Kitten in modo che esso esegua le sole analisi lessicale e sintattica (Capitolo 3) del sorgente da compilare;

`make semantical:` Compila il compilatore Kitten in modo da eseguire solo le analisi lessicale, sintattica e semantica (o type-checking, Capitolo 5) del sorgente da compilare;

make translate: Compila il compilatore Kitten in modo da eseguire solo le analisi lessicale, sintattica e semantica del sorgente da compilare e la generazione del codice intermedio Kitten bytecode (Capitolo 6);

make generatejb: Compila il compilatore Kitten in modo da eseguire solo le analisi lessicale, sintattica e semantica del sorgente da compilare, la generazione del codice intermedio Kitten bytecode e la sua trasformazione nel codice eseguibile Java bytecode (Capitolo 6);

make analyse: Compila il compilatore Kitten in modo da eseguire le analisi lessicale, sintattica e semantica del sorgente da compilare, la generazione del codice intermedio Kitten bytecode, alcune sue analisi statiche e la sua trasformazione nel codice eseguibile Java bytecode (Capitolo **??**).

Si noti che solo le due ultime modalità di compilazione generano un compilatore Kitten il quale a sua volta genera dei file eseguibili `.class` scritti in Java bytecode. Non c'è quindi da meravigliarsi se dopo un `make syntactical` il comando `./run` non funzioni o esegua una vecchia versione del programma che si vuole eseguire: a quel punto, il compilatore Kitten non genera più alcun file `.class` eseguibile. È necessario eseguire nuovamente un `make generatejb` oppure un `make analyse` per ottenere un compilatore Kitten che genera file `.class` eseguibili.

Altri comandi `make` effettuano operazioni di pulizia o documentazione del codice:

make clean: Pulisce le directory dell'installazione Kitten, eliminando file temporanei di `emacs`, i file `.class` di Java e file `dot` generati dentro la directory `testcases`;

make javadocs: Rigenera la documentazione JavaDoc del compilatore Kitten. Tale documentazione viene scritta dentro la directory `javadocs` ed è consultabile con un qualsiasi browser internet a partire dal file `index.html`.

Ci si abitui subito a consultare tale documentazione. Per esempio, si inserisca una url del tipo `file:///home/spoto/kitten_XXX/javadocs/index.html` dentro un browser internet. Ovviamente dovrete usare un nome di directory corrispondente al vostro punto di installazione del compilatore Kitten e alla struttura del vostro file system. Dovreste poter vedere l'elenco di tutti i package e di tutte le classi che compongono il compilatore Kitten, e consultare la documentazione dei loro campi e metodi. Usare tale documentazione facilita estremamente lo sviluppo di modifiche del compilatore Kitten, come per esempio eventuali progetti assegnati allo studente.

Adesso che abbiamo capito come installare e compilare il compilatore Kitten e come usarlo per compilare file scritti nel linguaggio Kitten, diamo un'occhiata più da vicino alla struttura e al funzionamento di tale linguaggio.

1.5 Comandi Kitten

Un *comando* è una porzione di codice Kitten la cui esecuzione può modificare lo stato dell'esecutore ma non fornisce alcun valore. Il concetto di *stato* va preso nel senso più generale possibile: esso include l'insieme delle variabili del programma, il loro tipo e valore, ma anche il contenuto del video. Per esempio, la Figura 1.1 contiene il comando `"miao\n".output()`, il cui effetto è di stampare su video la stringa `miao` seguita da un carattere di newline. Un altro esempio di comando Kitten è `int y := 0`, che *dichiara* una variabile di nome `y`, di tipo `int` e con valore iniziale pari a 0.

I comandi Kitten possono essere *composti*. Per esempio, se abbiamo due comandi c_1 e c_2, possiamo comporli sequenzialmente scrivendo $c_1; c_2$. Quello che otteniamo è ancora un comando, la cui esecuzione consiste nell'esecuzione di c_1 seguita dall'esecuzione di c_2. Per esempio, scrivendo `"hello".output(); " kitten".output()` otteniamo l'effetto di stampare su video la stringa `hello kitten`. Oltre a questa composizione sequenziale di comandi, Kitten fornisce delle forme standard di composizione di comandi che si trovano in tantissimi altri linguaggi. Per esempio, il *condizionale* `if (y >= 18) then` c_1 `else` c_2 è un comando che esegue c_1 se la variabile `y` contiene un valore maggiore o uguale a 18 ed esegue c_2 altrimenti. Se quindi scriviamo

```
if (y >= 18) then "man".output()
             else "kid".output()
```

otteniamo di stampare la stringa `man` se la variabile `y` contiene un valore maggiore o uguale a 18, e di stampare `kid` altrimenti. Si noti che, essendo in Kitten il punto e virgola un separatore di comandi, non dobbiamo inserirlo dopo il comando `"man".output()`.

Si faccia attenzione adesso al seguente codice:

```
if (y >= 18) then "old ".output(); "man".output()
             else "kid".output()
```

L'intenzione del programmatore era quella di stampare la stringa `old man` se la variabile `y` contiene un valore maggiore o uguale a 18, e di stampare `kid` altrimenti. Purtroppo il compilatore Kitten non ama tale codice, e segnala un errore di sintassi in corrispondenza all'`else`. Il motivo è che esso interpreta tale codice come un comando `if (y >= 18) then "old ".output()` seguito da un secondo comando `"man".output()` seguito a sua volta da uno stranissimo comando `else "kid".output()` e non trova neppure il punto e virgola che dovrebbe separare i comandi in Kitten. La giusta sintassi sarebbe stata invece la seguente:

```
if (y >= 18) then { "old ".output(); "man".output() }
             else "kid".output()
```

Questo codice viene accettato e compilato dal compilatore Kitten ed esegue esattamente quello che il programmatore aveva in mente. Le parentesi graffe sono un ulteriore costrutto di composizione di comandi. Se abbiamo un comando c, allora la notazione $\{c\}$ è ancora un comando, la cui esecuzione è semplicemente l'esecuzione di c. Tale costrutto permette in pratica di raggruppare

una sequenza di comandi per formare un comando unico di cui si specifica l'inizio e la fine, come nell'esempio precedente. Deve essere chiaro comunque che del codice fra parentesi graffe *è* un comando. Possiamo quindi dire che la parola chiave `then` è sempre seguita da uno *e un solo* comando (non da uno *o più* comandi). Similmente, in Kitten il corpo di un metodo o costruttore è sempre *un* comando. Per esempio, in Figura 1.2 il corpo del costruttore vuoto è il *comando vuoto* `{}`, la cui esecuzione lascia lo stato immutato. Il corpo del metodo `main` in Figura 1.1 è il comando `{ "miao\n".output() }`, che potrebbe essere semplificato in `"miao\n".output()`. In pratica, useremo le parentesi graffe solo se il corpo di un metodo o costruttore è così esteso che abbiamo bisogno di dividerlo in più comandi in sequenza, come accadrà in tantissimi casi.

Si noti che in C o Java le parentesi graffe sono ugualmente un costrutto per ottenere un comando composto, ma sono in alcuni casi obbligatorie dove Kitten potrebbe non richiederle, come all'inizio del corpo dei metodi e costruttori. Come abbiamo già detto, anche le regole di uso del punto e virgola sono diverse fra Kitten (dove esso è un *separatore* di comandi) e C e Java (dove esso è un *terminatore* di comandi). Queste differenze, unitamente all'obbligo in Kitten della parola chiave `then` nei condizionali, sono spesso all'origine di *misteriosi* messaggi di errore emessi dal compilatore Kitten e che lasciano perplessi non pochi studenti.

1.6 Valori Kitten

Abbiamo detto all'inizio della Sezione 1.5 che l'esecuzione di un comando non fornisce alcun *valore*. Ma cos'è un valore? Possiamo definirlo come un pezzo di informazione contenuto nella memoria del calcolatore. I valori possono essere creati, legati a variabili del programma, copiati, condivisi, modificati e confrontati.

Kitten gestisce i seguenti valori:

- valori *primitivi*, non creabili, né modificabili, né condivisibili:
  - i numeri interi: $\ldots, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \ldots$;
  - i numeri in virgola mobile a singola precisione: $3.14, -1.13, \ldots$;
  - i booleani *true* e *false*;
  - il riferimento *nil*;
- valori *non primitivi* o *riferimento*, creabili, modificabili e condivisibili:
  - gli *oggetti*, cioè delle zone di memoria divisibili in varie sotto-zone chiamate *campi* o *variabili d'istanza* dell'oggetto. I campi contengono a loro volta dei valori. La strutturazione in campi di un oggetto Kitten è descritta dalla *classe* dell'oggetto, che in Kitten viene specificata al momento della crezione dell'oggetto stesso e non è più mutabile; diremo che un oggetto è un'*istanza* della sua classe;
  - gli *array* o *vettori*, cioè delle zone di memoria divisibili in varie sotto-zone, chiamate *elementi* dell'array e indirizzabili tramite un riferimento numerico intero non negativo. Gli elementi di un array contengono a loro volta dei valori. La strutturazione

in elementi di un array, cioè il numero e il tipo degli elementi, viene specificata in Kitten al momento della sua creazione e dopo non è più mutabile.

I valori esistono solo a tempo di esecuzione. Si badi quindi a non confondere l'*espressione sintattica* `2` in Figura 1.2 con il suo *valore semantico* 2 che essa assume a tempo di esecuzione. Nel primo caso si tratta semplicemente di un carattere immesso dal programmatore nel testo del programma. Nel secondo caso si tratta di un'entità matematica, il numero intero 2 appunto. Questa distinzione è ancora più chiara per i valori non primitivi. Per esempio, l'espressione sintattica `new String()` in Figura 1.2 è solo una sequenza di caratteri digitati dal programmatore, il cui significato è di creare un oggetto di classe `String.kit` *a tempo di esecuzione*. In particolare, se la stessa espressione viene eseguita più volte, essa crea più oggetti diversi della classe `String.kit`.

In genere, in un linguaggio di programmazione si distinguono concetti relativi al momento della compilazione, come la sintassi delle espressioni, e concetti relativi al momento dell'esecuzione del programma, come il valore delle espressioni. Ritroveremo spesso questa distinzione in futuro. Spesso indicheremo come *statici* i primi e come *dinamici* i secondi.

I valori possono essere *legati* o *copiati* dentro a una variabile del programma. Per esempio, in Figura 1.2, l'oggetto creato dall'espressione `new String()` viene memorizzato dentro alla variabile `s`. Si noti che `s` non è una stringa. Essa è una variabile che *contiene* o *fa riferimento* a un oggetto stringa. Ciò nonostante, è convenzione comune dire che `s` *è* un oggetto quando si dovrebbe dire che `s` *contiene* un oggetto. Ci adegueremo anche noi a questo uso, ma deve essere chiara la distinzione fra variabile e suo valore.

Il comando `s := new String()` in Figura 1.2 è chiamato *assegnamento*. Si noti l'uso di `:=` al posto del solo = come si farebbe in C e Java. L'esecuzione di questo assegnamento consiste nell'assegnare alla variabile `s` un riferimento all'oggetto appena creato dall'espressione `new String()`. Si noti che non è l'oggetto che viene copiato, ma un suo riferimento, esattamente come in Java. Per cui un successivo assegnamento `s1 := s` avrebbe l'effetto di legare anche la variabile `s1` allo *stesso* oggetto a cui abbiamo appena legato la variabile `s`. In questo caso di oggetto ne esiste uno solo, raggiungibile sia tramite `s` che tramite `s1`. Diremo quindi che tale oggetto è *condiviso* fra `s` ed `s1`. Ogni modifica all'oggetto effettuata tramite `s` sarà automaticamente visibile anche tramite `s1`. La situazione è ben diversa per i valori primitivi, che vengono sempre copiati da un assegnamento. Per esempio, il comando `i := 3; i1 := i` ha l'effetto di legare sia `i` che `i1` allo stesso numero intero 3, ma una successiva modifica di `i` non influisce sul valore legato ad `i1`.

1.7 Espressioni Kitten

Un'*espressione* è un pezzo di codice Kitten la cui esecuzione, detta *valutazione*, può modificare lo stato dell'esecutore e, *inoltre*, fornisce un valore, chiamato appunto *valore dell'espressione*. Per esempio, la Figura 1.1 contiene l'espressione `"miao\n"`. Il suo valore è un oggetto di tipo

stringa che rappresenta la sequenza di caratteri `m-i-a-o` seguita da un carattere di newline. Un altro esempio è `s` in Figura 1.2. Essa è un'espressione il cui valore è il contenuto della variabile `s` in tale punto del programma. Si potrebbe pensare che, in effetti, le espressioni hanno un valore ma non modificano mai lo stato dell'esecutore. Questo è vero negli esempi fatti fin qui, ma in futuro vedremo che anche la chiamata di un metodo può essere un'espressione, purché il metodo non abbia `void` come tipo di ritorno. Il valore di una tale espressione è infatti il valore di ritorno del metodo. Un esempio è l'espressione `this.fib(n - 1)` in Figura 1.2. Dal momento che il corpo del metodo chiamato può contenere comandi arbitrari, dobbiamo ammettere che un'espressione Kitten possa modificare lo stato dell'esecutore. In particolare, diremo che un'espressione contiene un *side-effect* o *effetto di bordo* se la sua valutazione modifica lo stato dell'esecutore. Altrimenti essa non contiene side-effect. In Kitten l'unica espressione che può contenere side-effect è la chiamata di metodo. Altri linguaggi hanno altre espressioni con side-effect, come i preincrementi e postincrementi di variabili numeriche in linguaggi tipo C o Java: `i++`. Queste espressioni non esistono però in Kitten.

Anche le espressioni, come i comandi della Sezione 1.5, sono definite in maniera ricorsiva. Conseguentemente, possiamo *comporre* espressioni complesse a partire da espressioni più semplici. Per esempio, se abbiamo due espressioni e_1 ed e_2 allora possiamo comporle per formare l'espressione e_1`-` e_2 la cui valutazione valuta e_1, poi e_2 ed infine calcola la differenza del valore di e_1 meno quello di e_2. Tale differenza è il valore dell'espressione e_1`-` e_2. Un esempio è l'espressione `n - 1` in Figura 1.2.

I mondi delle espressioni e dei comandi non sono separati ma strettamente interdipendenti. Per esempio, il comando `"miao\n".output()` in Figura 1.1 è costruito a partire dall'espressione `"miao\n"`. In generale, potremmo dire che il comando *chiamata di metodo* si costruisce a partire da espressioni $e, e_1, \ldots, e_n$ con la sintassi e.`m`$(e_1, \ldots, e_n)$, dove `m` è il nome del metodo che si intende chiamare *sul* valore dell'espressione e *con parametri* pari al valore delle espressioni $e_1, \ldots, e_n$. Questo mostra che un comando Kitten può costruirsi a partire da espressioni Kitten. Il viceversa non accade in Kitten, ma può accadere in altri linguaggi. Per esempio, in C esiste l'*espressione virgola* (c,e) la cui esecuzione esegue il comando c, valuta l'espressione e e infine ritorna il valore di e.

1.8 Tipi Kitten

Kitten è un linguaggio di programmazione *tipato*. Questo vuol dire che i valori della Sezione 1.6 sono organizzati in gruppi detti appunto *tipi*. I tipi specificano le operazioni che su tali valori si possono effettuare. Per esempio, il tipo `int` raggruppa i valori interi di Kitten. Su un valore di tipo intero, Kitten permette di effettuare addizioni, sottrazioni, ecc. L'oggetto creato dall'espressione `new String()` in Figura 1.2 è di tipo `String`, per cui su di esso è possibile applicare tutti i metodi della Figura 1.3. Non è invece possibile applicare su di esso addizioni o sottrazioni.

I tipi Kitten corrispondono ai gruppi di valori che abbiamo descritto nella Sezione 1.6. In particolare, Kitten ha i tipi *primitivi* `int`, `float`, `boolean` e `nil`, nonché i tipi *non primitivi* corrispondenti a tutti i nomi delle classi che costituiscono il programma (per gli oggetti) e al tipo `array of` t dove t è il tipo degli elementi dell'array. L'insieme dei tipi Kitten è parzialmente

ordinato rispetto a una *relazione di sottotipaggio* $\leq$ che esprime la *compatibilità* fra tipi. Se $t_1 \leq t_2$ allora è possibile utilizzare un valore del tipo t_1 ogni volta che viene richiesto un valore di tipo t_2. Per esempio, è possibile assegnare un valore di tipo t_1 a una variabile dichiarata di tipo t_2, oppure passare tale valore come parametro a un metodo che si aspettava un valore di tipo t_2. La relazione $\leq$ è formalmente definita come la chiusura transitiva e riflessiva della seguente relazione:

$$\texttt{int} \leq \texttt{float}$$
$$\kappa_1 \leq \kappa_2 \quad \text{se la classe } \kappa_1 \text{ estende la classe } \kappa_2$$
$$\texttt{array of } t_1 \leq \texttt{array of } t_2 \quad \text{se } t_1 \leq t_2 \text{ e } t_1 \text{ non è un tipo primitivo}$$
$$\texttt{nil} \leq \kappa \quad \text{per ogni classe } \kappa$$
$$\texttt{nil} \leq \texttt{array of } t \quad \text{per ogni tipo } t$$
$$\texttt{array of } t \leq \texttt{Object} \quad \text{per ogni tipo } t.$$

Si noti che tale definizione implica che ogni array è un sottotipo della classe `Object` e che `array of int` $\leq$ `array of int` (per riflessività). Invece tale definizione implica che non è vero che `array of int` $\leq$ `array of float`, benché `int` $\leq$ `float`. La motivazione di questa non monotonia nella relazione di sottotipaggio degli array sarà chiara in seguito. Se $t_1 \leq t_2$ diremo che t_1 è un *sottotipo* di t_2 e che t_2 è un supertipo di t_1. Se $t_1 \neq t_2$ scriveremo $t_1 < t_2$ e diremo che t_1 è un sottotipo *stretto* di t_2 e che t_2 è un supertipo *stretto* di t_1.

Kitten è un linguaggio di programmazione a *tipaggio statico*. Questo vuol dire che a ciascuna espressione che figura in un programma Kitten viene associato un tipo *al momento della compilazione*. Le regole che specificano come questo tipo venga assegnato alle espressioni si chiamano *regole di tipaggio statico*. Lo strumento che effettua l'assegnazione di un tipo a ciascuna espressione di un programma si chiama *analizzatore semantico* o *type-checker* del linguaggio (Capitolo 5). Per esempio, l'analizzatore semantico di Kitten etichetta l'espressione `new String()` in Figura 1.2 con il tipo `String`. Esso inoltre etichetta l'espressione `this.fib(n - 1)` con il tipo `int`, poiché il tipo di ritorno del metodo `fib` è `int`. Molti linguaggi di programmazione di uso corrente hanno un tipaggio statico. Per esempio C, C++ e Java. Diverso è il caso di un linguaggio tipo Prolog e di alcuni dialetti del Basic, in cui viene assegnato un tipo alle espressioni ma solo *a tempo di esecuzione*. In tal caso, si parla di un linguaggio di programmazione a *tipaggio dinamico*. Si noti che il tipaggio dinamico permette a un'espressione di avere tipi diversi in tempi diversi, mentre il tipaggio statico deve assegnare uno e un solo tipo a un'espressione.

Cosa abbiamo guadagnato ad avere un linguaggio a tipaggio statico piuttosto che dinamico? Molto. In particolare, il tipaggio statico ci permette di eseguire l'analisi semantica del programma solo una volta a tempo di compilazione, piuttosto che in continuazione a tempo di esecuzione. Il programma sarà quindi più veloce. Non solo. Il tipaggio dinamico richiede di conoscere il tipo dei valori a tempo di esecuzione per potere ricostruire quello delle espressioni. Per esempio, qual è il tipo dell'espressione `y` a tempo di esecuzione? Occorre guardare cosa contiene `y` e capire di che tipo sia tale valore. Ma un valore è, dal punto di vista del computer, semplicemente una sequenza di bit. Due variabili `y` e `z` potrebbero contenere la stessa rappresentazione binaria ma avere valori diversi poiché una andava pensata (ovvero, era stata dichiarata) di tipo `int` e l'altra

di tipo `float`. Quello che manca è un'indicazione esplicita del tipo del valore. Ecco quindi che i linguaggi a tipaggio dinamico sono costretti ad associare ai valori un'etichetta che specifica il tipo del valore. Questa etichetta occupa spazio e rallenta l'esecuzione del programma. La coppia valore più etichetta è detta rappresentazione *boxed* di un valore. Il solo valore è detto rappresentazione *unboxed* di se stesso. Potremmo quindi dire che in un linguaggio a tipaggio statico è sufficiente utilizzare una rappresentazione unboxed dei valori, il che rende molto più semplice ed efficiente la sua implementazione rispetto a un linguaggio a tipaggio dinamico, in cui è obbligatoria la più pesante rappresentazione boxed. Va detto comunque che i linguaggio a tipaggio dinamico offrono maggiore flessibilità al programmatore e sono quindi spesso preferiti da chi lavora nell'ambito dell'intelligenza artificiale. Infine, va osservato che nei linguaggi a oggetti si è comunque obbligati a usare una rappresentazione boxed per la maggior parte degli oggetti (normalmente, per tutti gli oggetti), anche se il linguaggio ha un tipaggio statico, a causa della presenza di chiamate virtuali con late-binding e di cast controllati a tempo di esecuzione. La rappresentazione unboxed è quindi limitata ai soli valori primitivi.

Kitten è un linguaggio di programmazione *fortemente tipato*. Questo significa in primo luogo che esso è tipato. Ma significa anche che il tipo assegnato (a tempo di compilazione, nel caso di Kitten) alle espressioni dall'analizzatore semantico è un supertipo di quello del valore delle stesse espressioni a tempo di esecuzione. In altre parole, l'analizzatore semantico ha etichettato *bene*, senza *sbagliare*, le espressioni del programma, in modo tale che il tipo scelto è corretto (un supertipo, appunto) rispetto a quello che poi avranno tali espressioni quando andremo ad eseguire il programma. Per esempio, il fatto che l'espressione `new String()` in Figura 1.2 sia stata etichettata con il tipo `String` è consistente con il fatto che, a tempo di esecuzione, tale espressione avrà un valore di tipo `String`. Come Kitten, anche il linguaggio Java è fortemente tipato. Mentre non sono fortemente tipati C, C++ o C#, poiché è possibile camuffare il tipo delle espressioni tramite cast non controllati, ed è anche possibile spacciare per oggetti puntatori a memoria forgiati a partire da valori interi, o infine accedere a zone di memoria oltre i limiti di un array senza garanzia che questo blocchi l'esecuzione del programma e forgiando quindi *valori* del tipo dell'array a partire da configurazioni causali di bit.

Cosa abbiamo guadagnato dalla scelta di avere un linguaggio *fortemente* tipato piuttosto che semplicemente tipato? Anche in questo caso abbiamo guadagnato molto. In particolare, abbiamo ottenuto la possibilità di garantire, a tempo di compilazione, che tutta una serie di proprietà relative ai tipi dei valori delle espressioni saranno sicuramente vere a tempo di esecuzione. Non servirà quindi verificarle a tempo di esecuzione. Per esempio, si consideri l'espressione `s` in Figura 1.2. Essa riceve dall'analizzatore semantico il tipo `String`. Poiché Kitten è fortemente tipato, sappiamo che a tempo di esecuzione dentro la variabile `s` troveremo *realmente* un valore di tipo `String`. Conseguentemente, non occorre verificare, a tempo di esecuzione, che l'oggetto contenuto dentro `s` abbia effettivamente un metodo chiamato `input`, perché tale controllo è superfluo, avendo `String` un metodo chiamato `input`. In altre parole, un linguaggio a tipaggio forte ci permette di spostare a tempo di compilazione un gran numero di controlli sulla consistenza di quello che il programmatore ha scritto. Tali controlli devono invece essere effettuati a tempo di esecuzione nel caso di un linguaggio a tipaggio debole (cioè non forte). Da un punto di vista di verifica del software, possiamo dire che un linguaggio a tipaggio forte permette all'analizzatore semantico di *dimostrare* a tempo di compilazione che nulla *andrà storto* a tempo di

esecuzione, relativamente a una larga classe di errori di tipo che include l'esistenza dei metodi al momento della loro chiamata, la correttezza dei valori passati ai metodi, la consistenza dei valori assegnati alle variabili, ecc. Ma non include purtroppo proprietà come la correttezza dei cast, la non-nullness dei ricevitori delle chiamate di metodo e degli accessi ai campi e la legalità degli indici per l'accesso agli array. Tali proprietà vengono normalmente verificate a tempo di esecuzione anche nei linguaggi a tipaggio forte.

L'analizzatore semantico, che assegna un tipo statico alle espressioni e garantisce la correttezza di comandi ed espressioni, è spesso estremamente severo, al punto da vietare cose che apparentemente sono giudicate *corrette* da molti programmatori. Per esempio, se `y` è una variabile di tipo intero ed `s` è una variabile di tipo `String`, esso considera come scorretto il comando

```
if (y >= y) then y := 2 else s := 3
```

poiché ritiene illegale assegnare as `s` il valore intero 3. Un programmatore potrebbe forse osservare che questa scelta è troppo pessimistica, dal momento che la guardia del condizionale sarà sempre vera a tempo di esecuzione e quindi il ramo `else` del condizionale non verrà mai eseguito. Osserviamo però che non è in genere possibile determinare se la guardia di un condizionale è sicuramente vera o sicuramente falsa: si tratta di un problema *indecidibile*, come la maggior parte dei problemi *interessanti* dei programmi per calcolatore. Non possiamo pretendere quindi che l'analizzatore semantico decida un problema indecibile, né sembra una buona idea quella di considerare alcuni casi speciali (come qui `y >= y`), dal momento che corrispondono a usi rari e spesso errati delle guardie. È meglio continuare ad accettare il giudizio di un analizzatore semantico forse un po' severo, ma sicuramente corretto.

1.9 Classi e campi Kitten

Abbiamo detto che un oggetto è una zona di memoria divisa in sotto-zone dette *campi*. Tali campi sono identificati da un nome che è dato al momento della dichiarazione della classe di cui l'oggetto è un'istanza. Si consideri per esempio la classe `Led.kit` in Figura 1.4. Tale classe implementa una lampadina che può essere accesa o spenta tramite i metodi `on()` e `off()` e di cui si può controllare lo stato di accensione tramite i metodi `isOn()` e `isOff()`. Tutti gli oggetti di classe `Led` contengono al loro interno una zona di memoria etichettata come `state` e che contiene un valore di tipo booleano. Ogni oggetto avrà la sua zona di memoria `state` e la modifica del campo `state` di un oggetto non riguarda il campo `state` di un altro oggetto. Si noti che la dichiarazione di questo campo richiede la parola chiave `field` che è invece sottintesa in C++ o Java. Si noti inoltre che in Kitten i campi sono tutti implicitamente pubblici, cioè accessibili dal codice di qualsiasi classe, senza alcuna restrizione di visibilità.

L'accesso a un campo avviene tramite la *notazione punto*: *e*`.f` dove `f` è il nome del campo ed *e* è l'espressione il cui valore deve essere un oggetto che ha un campo di nome `f`. Tale valore è detto *ricevitore* dell'accesso al campo. Se il ricevitore è *nil* si avrà un errore a tempo di esecuzione. Si noti che l'espressione *e* non può essere lasciata sottintesa quando essa è `this`,

```
class Led {
  field boolean state

  constructor() {}

  method void on()
    this.state := true

  method void off()
    this.state := false

  method boolean isOn()
    return this.state

  method boolean isOff()
    return !this.state
}
```

Figura 1.4: Una classe Kitten che implementa una lampadina.

come invece si fa in C++ o Java. Per esempio, sarebbe stato un errore in Figura 1.4 scrivere `state` al posto di `this.state`, perché sarebbe stato interpretato dal compilatore Kitten come una *variabile* di nome `state` e non come il *campo* `state` di `this`. Ma nessuna variabile di nome `state` è stata dichiarata in Figura 1.4.

La Figura 1.5 mostra una rappresentazione di un oggetto di tipo `Led`, supponendo che esso sia allocato in memoria a partire dalla locazione 1000. Tale rappresentazione è detta *stato* dell'oggetto. Si noti che la rappresentazione è boxed (Sezione 1.8). Potremmo assumere che l'etichetta con il nome della classe dell'oggetto sia una vera e propria stringa. In realtà si può tranquillamente usare un identificatore numerico unico di 32 bit al posto del nome della classe, risparmiando significativamente in occupazione di memoria rispetto all'uso di una stringa. Per questo motivo in Figura 1.5 abbiamo assunto che l'etichetta della classe occupi solo 4 byte. Si noti che il valore booleano del campo `state` occupa anch'esso 4 byte, sebbene un singolo bit sarebbe stato sufficiente. Questa scelta, se da una parte porta a un consumo di memoria, dall'altra permette di uniformare la dimensione dei campi a 4 byte per ogni campo, semplificando lo sviluppo del compilatore ma anche l'esecuzione del codice, visto che molti processori hanno una maggiore facilità di accesso a indirizzi di memoria che sono multipli di 4 (*allineamento* sulla parola), e permettono la lettura veloce di 4 byte alla volta.

Si noti che il nome del campo, `state`, non è presente in Figura 1.5. Come si fa quindi ad accedere al campo `state` di un oggetto come quello mostrato in tale figura? La risposta è semplice. Avendo gli oggetti di tipo `Led` un unico campo di nome `state`, il suo contenuto si trova subito dopo l'etichetta con il nome della classe. Nel nostro caso, subito dopo l'etichetta `Led`. L'etichetta `state` quindi non serve. Basta conoscere lo *spostamento* o *offset* a cui si

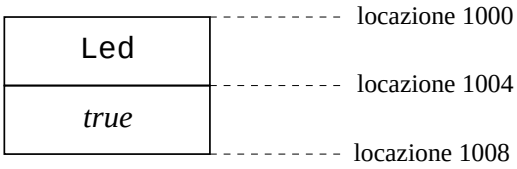


Figura 1.5: La rappresentazione (boxed) o stato di un oggetto di classe `Led`.

trova il valore del campo `state` a partire dall'inizio o *base* dell'oggetto. Nel caso del campo `state` di un oggetto di classe `Led`, l'offset è 4 byte. L'accesso tramite offset permette di non sprecare memoria per memorizzare il nome dei campi dentro agli oggetti, nonché di velocizzarne l'accesso dal momento che basta un'addizione dell'offset alla base dell'oggetto per indirizzare il campo, piuttosto che una lenta ricerca della stringa `state` all'interno dell'oggetto. Questo è un altro esempio del vantaggio che traiamo dall'uso di un linguaggio a tipizzazione statica. Per accedere a e`.state`, se il compilatore sa che l'espressione e ha tipo `Led` allora esso può generare del codice che esegue un'addizione di 4 byte dalla base del valore di e. Questo non sarebbe possibile se il tipaggio fosse dinamico, nel qual caso non sapremmo qual è il tipo di e se non al momento dell'esecuzione e non potremmo quindi calcolare alcun offset al momento della compilazione.

1.10 Metodi Kitten

Un *metodo* è una porzione di codice etichettata con un nome, la cui esecuzione richiede di fornire i valori, detti *parametri attuali*, di alcuni *parametri formali* e può restituire un valore detto *di ritorno*. La *dichiarazione* di un metodo Kitten richiede di specificare il tipo dei parametri formali e del valore di ritorno, come abbiamo fatto in Figura 1.2 per i metodi `fib` e `main`. La *chiamata* di un metodo si effettua con la *notazione punto* e`.m(`$e_1, \ldots, e_n$`)`. L'effetto è di valutare le espressioni e, e_1, ..., e_n (i parametri attuali) e di legarne i valori a `this` e ai parametri formali del metodo, che viene quindi eseguito. Se esso ritorna un valore tramite il comando `return`, quello è il valore di ritorno del metodo.

Si noti che in Kitten i metodi si chiamano *su* un valore contenuto in un'espressione e che sta alla sinistra del punto. Tale valore dovrà essere un oggetto (quindi diverso da *nil*) di una classe in cui è dichiarato un metodo di nome `m` con parametri formali compatibili con quelli forniti dalla chiamata, o di una classe da cui tale metodo è rintracciabile risalendo la catena delle superclassi. Il valore dell'espressione e è detto *ricevitore* della chiamata di metodo. Se ne evince che in Kitten il ricevitore è sempre un oggetto, ad eccezione del metodo `main` che viene invocato implicitamente sulla classe dalla macchina virtuale Java (Sezione 1.1). In altri linguaggi, tipo Java, è possibile invece usare una classe come ricevitore, nel caso in cui il metodo sia dichiarato come `static`, nonché gli array, che in Java hanno tutti e soli i metodi ereditati da `java.lang.Object`.

La classe del ricevitore determina l'implemetazione del metodo che viene eseguita da una chiamata di metodo. Questo è evidente se utilizziamo una caratteristica dei linguaggi a oggetti,

```
class S {
  constructor () {}

  method String toString()
    return "I'm an S;"
}

class B extends S {
  constructor () {}

  method String toString()
    return "I'm a B;"
}

class A extends S {
  constructor () {}

  method String toString()
    return "I'm an A;"
}
```

```
class Virtual {
  constructor () {}

  method void main() {
    Virtual v := new Virtual();
    v.print(new A());
    v.print(new B());
    v.print(new S())
  }

  method void print(S s)
   s.toString().output()
}
```

Figura 1.6: Un esempio di definizione di classi per estensione e di late-binding.

cioè la possibilità di definire una classe *a partire* o *estendendo* un'altra classe. Si consideri per esempio la Figura 1.6. Diremo che la classe `S` è una *superclasse* di `A` e `B`, che sono invece sue *sottoclassi*. Il metodo `main` della classe `Virtual` crea tre oggetti, uno di tipo `A`, uno di tipo `B` e uno di tipo `S`, e li passa uno dopo l'altro a un metodo `print` che si aspetta un parametro di tipo `S`. Questo è possibile poiché $\texttt{A} \leq \texttt{S}$ e $\texttt{B} \leq \texttt{S}$. Il metodo `print` chiama il metodo `toString()` su tali oggetti e ne stampa il valore di ritorno. Il risultato è

```
I'm an A;I'm a B;I'm an S;
```

Questo significa che la chiamata `s.toString()` ha *selezionato* di volta in volta un'implementazione diversa dello stesso metodo `toString` sulla base della classe dell'oggetto `s`, che è `A` la prima volta che `print` viene chiamato, è `B` la seconda ed è `S` la terza. Questa selezione viene effettuata a tempo di esecuzione, guardando l'etichetta contenuta nell'oggetto ricevitore contenuto in `s`, la quale ne specifica la classe. È questo uno dei motivi per cui nei linguaggi a oggetti la rappresentazione degli oggetti è normalmente boxed (Sezione 1.6).

Il fatto che l'implementazione di un metodo non è nota a tempo di compilazione ma solo a tempo di esecuzione è detta *legame ritardato* fra chiamante e chiamato, o *late-binding*. Essa è una caratteristica dei linguaggi a oggetti e fornisce la base dell'estendibilità del software a oggetti, che è una delle motivazioni per cui i linguaggi a oggetti sono stati creati. Va detto che il late-binding è inerentemente lento, poiché richiede di accedere all'etichetta che identifica la

```
class Arrays {
  method void main() {
    array of S arr := new S[10];
    for (int i := 0; i < 10; i := i + 1)
      if (i - (i / 3) * 3 = 0) then arr[i] := new A()
      else if (i - (i / 3) * 3 = 1) then arr[i] := new B()
      else arr[i] := new S();

    int i := 0;
    while (i < 10) {
      arr[i].toString().concat("\n").output();
      i := i + 1
    }
  }
}
```

Figura 1.7: La classe `Arrays.kit` che crea, inizializza e stampa un array.

classe del ricevitore e quindi di ricercare l'implementazione del metodo all'interno di tale classe. Una chiamata diretta, con *legame anticipato* a tempo di compilazione, o *early-binding*, come in C, è nettamente più veloce ma meno flessibile e non estendibile.

Si noti che in Kitten il late-binding avviene solo per i metodi e non per i campi, esattamente come in Java. La ridefinizione di un campo ha quindi l'effetto di dichiarare un *altro* campo con lo stesso nome del campo ridefinito. I due campi vengono distinti sulla base del tipo *statico* del ricevitore.

1.11 Alcuni esempi conclusivi

La Figura 1.7 mostra un programma che crea un array di `S` (Figura 1.6), inizializza i suoi elementi con un ciclo `for` e quindi li stampa con un ciclo `while`. Si noti che questi due costrutti iterativi sono molto simili a quelli di C, C++ o Java. Non sono però disponibili le istruzioni `break` e `continue` fornite da questi ultimi linguaggi. Si noti che è possibile ridefinire una variabile: la nuova dichiarazione nasconde la precedente e può avere un tipo diverso da quello della prima dichiarazione. Osserviamo inoltre che agli elementi di un array è possibile assegnare qualsiasi valore compatibile con il tipo di dichiarazione di tali elementi. L'esecuzione del programma in Figura 1.7 stampa:

```
I'm an A;
I'm a B;
I'm an S;
I'm an A;
I'm a B;
```

```
I'm an S;
I'm an A;
I'm a B;
I'm an S;
I'm an A;
```

ancora una volta grazie al late-binding della chiamata al metodo `toString`.

La Figura 1.8 mostra una classe Kitten che compone sette led al fine di formare un display capace di rappresentare le dieci cifre (la versione completa di questa classe è disponibile nella directory `testcases` della distribuzione di Kitten). Il costruttore crea i led. Il metodo `showDigit` accende e spegne i led in modo da visualizzare la cifra richiesta. Il metodo `increment` incrementa di uno la cifra rappresentata dal display; si noti l'uso di un `return` che restituisce un'uguaglianza fra due espressioni, il che è corretto dal momento che l'uguaglianza fra due espressioni ha tipo booleano e può quindi essere restituita da un metodo il cui tipo di ritorno è stato dichiarato come `boolean`. Infine, il metodo `toString` restituisce una rappresentazione del display sotto forma di stringa. La Figura 1.9, infine, mostra una classe che crea due cifre, le incrementa e le stampa a video. Il risultato è il seguente e mostra come ciascuno dei due oggetti `Digit` abbia un diverso campo `digit`, indipendente da quello dell'altro oggetto:

```
By incrementing
 _
 _|
 _|

we get:

|_|
  |

without carry

By incrementing
 _
|_|
 _|

we get:
 _
| |
|_|

with carry
```

```
class Digit {
  field Led led1   field Led led2   field Led led3
  field Led led4   field Led led5   field Led led6
  field Led led7   field int digit

  constructor() {
    this.led1 := new Led(); this.led2 := new Led();
    this.led3 := new Led(); this.led4 := new Led();
    this.led5 := new Led(); this.led6 := new Led();
    this.led7 := new Led(); this.showDigit(0)
  }

  method void showDigit(int digit) {
    this.digit := digit;
    if (this.digit = 0) then {
      this.led1.on(); this.led2.on();
      this.led3.on(); this.led4.off();
      this.led5.on(); this.led6.on(); this.led7.on()
    } else if (this.digit = 1) then ...
  }

  method boolean increment() {
    this.digit := this.digit + 1;
    if (this.digit = 10) then this.digit := 0;
    this.showDigit(this.digit);
    return this.digit = 0
  }

  method String toString() {
    String result := "";
    if (this.led1.isOn()) then result := result.concat(" _\n")
                          else result := result.concat("  \n");
    ...
    return result
  }
}
```

Figura 1.8: La classe `Digit.kit`, che rappresenta le dieci cifre con un display di led.

```
class Increment {
  method void main() {
    Digit digit1 := new Digit();
    Digit digit2 := new Digit();
    boolean carry := false;

    digit1.showDigit(3);
    digit2.showDigit(9);

    "By incrementing\n".concat(digit1.toString())
       .concat("\n\nwe get:\n").output();

    carry := digit1.increment();
    digit1.toString().concat("\n\n").output();
    if (carry) then "with carry\n\n".output()
               else "without carry\n\n".output();

    "By incrementing\n".concat(digit2.toString())
       .concat("\n\nwe get:\n").output();

    carry := digit2.increment();
    digit2.toString().concat("\n\n").output();
    if (carry) then "with carry\n".output()
               else "without carry\n".output()
  }
}
```

Figura 1.9: La classe `Increment.kit`, che crea due cifre, le incrementa e le stampa.

Esercizio 1. Si esamini la classe `List.kit` fornita con Kitten, nella directory `testcases`. Si cerchi di comprendere il funzionamento dei metodi e si provi ad aggiungerne di nuovi.

Esercizio 2. Si scriva una classe `Scramble.kit` con un costruttore che riceve una stringa e un metodo `scramble` che stampa tutte le permutazioni della stringa fornita al momento della costruzione dell'oggetto.

Esercizio 3. Si scriva una classe Kitten `Sort.kit` con un costruttore che riceve come parametro un array di interi, un metodo `toString` che restituisce una stringa e vari metodi void e senza parametri, che implementano ciascuno un differente algoritmo di ordinamento degli array.

Esercizio 4. Si scriva una classe simile a quella dell'esercizio 3 ma con un unico metodo di ordinamento, che non effettua alcuna operazione sull'array. Si definiscano quindi delle sottoclassi, ciascuna con una diversa implementazione del metodo di ordinamento. Si scriva un main di prova. Notate l'importanza delle classi `abstract` di Java, purtroppo non disponibili in Kitten?

Capitolo 2

Analisi Lessicale

L'analisi lessicale scompone un sorgente Kitten in una sequenza di *token*. Ogni token rappresenta un insieme di stringhe che hanno lo stesso *ruolo* all'interno del sorgente Kitten. Per esempio, dei token identificano le parole chiave del linguaggio, un altro gli identificatori, un altro le costanti numeriche e così via. Identificare il ruolo delle parole che compongono un sorgente Kitten è essenziale per potere poi ricostruire la struttura sintattica del codice (Capitolo 3). In questo capitolo vedremo come implementare un analizzatore lessicale usando degli automi a stati finiti e delle espressioni regolari e come automatizzare la creazione di un analizzatore lessicale che riconosce un dato insieme di token, specificato da una lista di espressioni regolari. Useremo a tal fine il generatore JLex di analizzatori lessicali, che è una versione Java del programma lex [6] inizialmente sviluppato per il linguaggio C.

2.1 I token Kitten

Abbiamo detto che un token rappresenta un insieme di stringhe. Per esempio, il token `THEN` rappresenta l'insieme di stringhe {`then`}, mentre il token `ID` rappresenta l'insieme (potenzialmente infinito) delle stringhe che sono *identificatori* Kitten, cioè nomi di variabili, classi, campi e metodi. I token che rappresentano più di una stringa, come `ID`, hanno associato un *valore lessicale*, cioè la specifica stringa che essi rappresentano, caso per caso. Si consideri per esempio la classe `Led.kit` in Figura 1.4. Il risultato della sua analisi lessicale è in Figura 2.1. Si noti come le parole chiave del linguaggio, come `class`, `field`, `void`, `boolean`, sono rappresentate da un token specifico. Gli identificatori sono invece rappresentati dall'unico token `ID` con associato

```
CLASS from 0 to 4
ID(Led) from 6 to 8
LBRACE from 10 to 10
FIELD from 14 to 18
BOOLEAN from 20 to 26
ID(state) from 28 to 32
CONSTRUCTOR from 37 to 47
LPAREN from 48 to 48
RPAREN from 49 to 49
LBRACE from 51 to 51
RBRACE from 52 to 52
METHOD from 57 to 62
VOID from 64 to 67
ID(on) from 69 to 70
LPAREN from 71 to 71
RPAREN from 72 to 72
ID(this) from 78 to 81
DOT from 82 to 82
ID(state) from 83 to 87
ASSIGN from 89 to 90
TRUE from 92 to 95
METHOD from 100 to 105
VOID from 107 to 110
ID(off) from 112 to 114
LPAREN from 115 to 115
RPAREN from 116 to 116
ID(this) from 122 to 125
DOT from 126 to 126
ID(state) from 127 to 131
ASSIGN from 133 to 134
FALSE from 136 to 140
METHOD from 145 to 150
BOOLEAN from 152 to 158
ID(isOn) from 160 to 163
LPAREN from 164 to 164
RPAREN from 165 to 165
RETURN from 171 to 176
ID(this) from 178 to 181
DOT from 182 to 182
ID(state) from 183 to 187
METHOD from 192 to 197
BOOLEAN from 199 to 205
ID(isOff) from 207 to 211
LPAREN from 212 to 212
RPAREN from 213 to 213
RETURN from 219 to 224
NOT from 226 to 226
ID(this) from 227 to 230
DOT from 231 to 231
ID(state) from 232 to 236
RBRACE from 238 to 238
EOF from 239 to 239
```

Figura 2.1: Il risultato dell'analisi lessicale della classe in Figura 1.4.

un valore lessicale, cioè la stringa dell'identificatore. Si sarebbe potuto rappresentare anche le parole chiave come identificatori con associato un valore lessicale. Per esempio, si poteva usare `ID(method)` piuttosto che `METHOD`. La scelta dei token è in parte arbitraria, ma è in effetti pensata per semplificare la successiva ricostruzione della struttura grammaticale del testo, nella fase di analisi sintattica (Capitolo 3). A quel punto, sarà più semplice avere a che fare con `METHOD` piuttosto che con `ID(method)`. Si noti, in Figura 2.1, come le parentesi tonde siano rappresentate da due appositi token (`LPAREN` ed `RPAREN`), così come le parentesi graffe (`LBRACE` ed `RBRACE`). Lo stesso accade per le parentesi quadre (`LBRACK` ed `RBRACK`), non mostrate in figura. Alla fine è presente il token fittizio `EOF` che segnala la fine del file sorgente. Si noti che spazi, tabulazioni e commenti sono assenti in Figura 2.1. Essi vengono infatti scartati dall'analizzatore lessicale (Sezione 2.9).

2.2 Token come espressioni regolari

La specifica dei token di un linguaggio, come Kitten, richiede l'enumerazione dei token e la descrizione dell'insieme di stringhe che ciascun token rappresenta. In prima approssimazione, questi insiemi devono essere disgiunti, in modo che una data stringa possa appartenere ad al più un token. La descrizione delle stringhe rappresentate da un token potrebbe essere fornita in maniera informale, per esempio in italiano o inglese. Questa scelta avrebbe come conseguenza negativa la difficile automatizzazione della generazione dell'analizzatore lessicale, nonché la possibile ambiguità nella specifica dei token. Decidiamo quindi di usare un linguaggio formale per specificare i token del linguaggio. Tale linguaggio sarà quello delle *espressioni regolari*.

Un alfabeto specifica l'insieme dei caratteri con cui possiamo comporre i nostri programmi. Per esempio, potremmo supporre che l'alfabeto di Kitten siano i caratteri presenti sulla tastiera del calcolatore, o l'insieme dei caratteri unicode.

Definizione 1 (Alfabeto)**.** Un *alfabeto* Λ è un insieme finito di elementi, detti *caratteri*.

Un'espressione regolare è un elemento del seguente insieme, costruito a partire da un alfabeto.

Definizione 2 (Espressione Regolare)**.** L'insieme delle *espressioni regolari* su un alfabeto Λ è il più piccolo insieme $\mathcal{R}$ tale che

- $\emptyset \in \mathcal{R}$ (l'insieme vuoto è un'espressione regolare)
- $\varepsilon \in \mathcal{R}$ (la stringa vuota è un'espressione regolare)
- $\Lambda \subseteq \mathcal{R}$ (ogni carattere è un'espressione regolare)
- se $r_1, r_2 \in \mathcal{R}$ allora $r_1 r_2 \in \mathcal{R}$ (l'insieme delle espressioni regolari è chiuso per *sequenza* o *concatenazione*)
- se $r_1, r_2 \in \mathcal{R}$ allora $r_1|r_2 \in \mathcal{R}$ (l'insieme delle espressioni regolari è chiuso per *alternanza*)
- se $r \in \mathcal{R}$ allora $r^* \in \mathcal{R}$ (l'insieme delle espressioni regolari è chiuso per *iterazione*).

Se, per esempio, $\Lambda = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, allora $\mathtt{a} \in \mathcal{R}$, ma anche $\mathtt{abc} \in \mathcal{R}$, nonché $\mathtt{a|b|abc} \in \mathcal{R}$ ed $\mathtt{ab}^* \in \mathcal{R}$. Assumeremo di potere usare delle parentesi tonde nelle espressioni regolari, al fine di chiarire la loro struttura sintattica. Per esempio, scriveremo $\mathtt{a|b|(abc)}$ per distingure tale espressione regolare da $\mathtt{(a|b|a)bc}$ e scriveremo $\mathtt{(ab)}^*$ per distinguere tale espressione regolare da $\mathtt{a(b}^*\mathtt{)}$. Assumeremo che $*$ abbia massima priorità, per cui, in assenza di parentesi, $\mathtt{ab}^*$ va inteso come $\mathtt{a(b}^*\mathtt{)}$. Va comunque osservato che le parentesi non fanno strettamente parte del linguaggio delle espressioni regolari, ma servono solo a evidenziare la struttura della loro definizione induttiva. Ecco perché le parentesi non figurano nella Definizione 2.

Definita la *sintassi* delle espressioni regolari, passiamo a definire la loro *semantica*. Dal momento che i token rappresentano insiemi di stringhe e che vogliamo usare le espressioni regolari per specificare i token, sembra sensato che la semantica o significato di una espressione regolare sia un insieme di stringhe, ovvero un *linguaggio*.

Definizione 3 (Linguaggio)**.** Un *linguaggio* su un alfabeto Λ è un qualsiasi sottoinsieme delle stringhe finite ottenibili a partire dai caratteri di Λ. Tale insieme di stringhe è tradizionalmente indicato come Λ^*.

Per esempio, l'insieme $\{\texttt{then}\}$ è un linguaggio sull'alfabeto inglese, formato da un'unica stringa. L'insieme $\{\texttt{a}, \texttt{aa}, \texttt{aaa}, \ldots\}$ è un linguaggio sull'alfabeto $\{\texttt{a}, \texttt{b}\}$ formato da tutte e sole le stringhe formate da una o più $\texttt{a}$. Si noti che $\emptyset$ è un linguaggio su qualsiasi alfabeto, formato dall'insieme finito di stringhe. Anche $\{\varepsilon\}$ è un linguaggio su qualsiasi alfabeto, formato dalla stringa vuota. Si noti che $\emptyset \neq \{\varepsilon\}$.

Definizione 4 (Linguaggio di un'Espressione Regolare)**.** Data un'espressione regolare r sull'alfabeto Λ, la sua *semantica* $\mathcal{L}(r)$ è il linguaggio su Λ definito tramite le seguenti regole:

- $\mathcal{L}(\emptyset) = \emptyset$
- $\mathcal{L}(\varepsilon) = \{\varepsilon\}$
- $\mathcal{L}(a) = \{a\}$ per ogni $a \in \Lambda$
- $\mathcal{L}(r_1 r_2) = \{s_1 s_2 \mid s_1 \in \mathcal{L}(r_1) \text{ ed } s_2 \in \mathcal{L}(r_2)\}$
- $\mathcal{L}(r_1 | r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$
- $\mathcal{L}(r^*) = \{s_1 \cdots s_n \mid n \geq 0 \text{ ed } s_i \in \mathcal{L}(r) \text{ per ogni } 0 \leq i \leq n\}$.

Si noti che nella semantica di $\mathcal{L}(r^*)$ si ammette che $n = 0$, nel qual caso $s_1 \cdots s_n = \varepsilon$. Concludiamo che $\varepsilon \in \mathcal{L}(r^*)$ per ogni espressione regolare r.

Diremo spesso che $\mathcal{L}(r)$ è il linguaggio *denotato* dall'espressione regolare r o, più semplicemente, il *linguaggio dell'espressione* r.

Si consideri per esempio l'espressione regolare $\texttt{then}$ sull'alfabeto inglese. Essa denota il linguaggio

$$\mathcal{L}(\texttt{then}) = \{s_1 s_2 s_3 s_4 \mid s_1 \in \mathcal{L}(\texttt{t}),\ s_2 \in \mathcal{L}(\texttt{h}),\ s_3 \in \mathcal{L}(\texttt{e}) \text{ ed } s_4 \in \mathcal{L}(\texttt{n})\} = \{\texttt{then}\}.$$

L'espressione regolare $\texttt{aa}^*$ denota il linguaggio

$$\begin{aligned}\mathcal{L}(\texttt{aa}^*) &= \{s_1 s_2 \mid s_1 \in \mathcal{L}(\texttt{a}) \text{ ed } s_2 \in \mathcal{L}(\texttt{a}^*)\} \\ &= \{s_1 s_2 \mid s_1 \in \{\texttt{a}\} \text{ ed } s_2 = \underbrace{\texttt{a} \cdots \texttt{a}}_{n} \text{ con } n \geq 0\} \\ &= \{\underbrace{\texttt{a} \cdots \texttt{a}}_{n} \mid n \geq 1\}.\end{aligned}$$

Possiamo similmente definire l'espressione regolare che denota l'insieme degli *identificatori*, cioè delle sequenze non vuote di caratteri alfabetici, come $\alpha\alpha^*$, dove $\alpha = \texttt{a}|\texttt{b}|\texttt{c}|\texttt{d}|\cdots|\texttt{w}|\texttt{x}|\texttt{y}|\texttt{z}$. Si noti che i linguaggi di programmazione, come Kitten, usano una definizione più complessa di *identificatore*: una sequenza non vuota di caratteri, minuscoli o maiuscoli, di cifre e del carattere $\_$, che cominci però con un carattere alfabetico. Per poter definire tale nozione di identificatore, usiamo delle *abbreviazioni*.

Definizione 5 (Abbreviazioni)**.** Le seguenti sintassi sono abbreviazioni delle corrispondenti espressioni regolari:

- $\alpha?$ è un'abbreviazione per $\alpha|\varepsilon$
- α^+ è un'abbreviazione per $\alpha\alpha^*$
- se l'alfabeto è totalmente ordinato, allora $[\mathtt{a}-\mathtt{z}]$ è un'abbreviazione per $\mathtt{a}|\mathtt{b}|\mathtt{c}|\cdots|\mathtt{x}|\mathtt{y}|\mathtt{z}$ dove $\mathtt{b}|\mathtt{c}|\cdots|\mathtt{x}|\mathtt{y}$ è l'insieme dei caratteri compresi fra a e z. Questa abbreviazione viene estesa a intervalli multipli, come in $[\mathtt{a}-\mathtt{z}\mathtt{A}-\mathtt{Z}]$.

A questo punto possiamo definire gli identificatori come il linguaggio denotato dall'espressione regolare

$$\mathtt{ID} = [\mathtt{a}-\mathtt{z}\mathtt{A}-\mathtt{Z}][\mathtt{a}-\mathtt{z}\mathtt{A}-\mathtt{Z}\mathtt{0}-\mathtt{9}\_]^*$$

Compreso il linguaggio delle espressioni regolari, possiamo immaginare di specificare i token di un linguaggio di programmazione con un insieme di espressioni regolari, ognuna etichettata con il token che essa rappresenta. Ci sono però delle problematiche inerenti a questa semplice idea:

Coincidenza più lunga. Può accadere che la stessa stringa sia riconoscibile come un unico token o come due token consecutivi. Per esempio, la stringa `ciao` può essere riconosciuta come un'istanza del token `ID` per gli identificari ma anche come due istanze contigue del token `ID`: l'identificatore `ci` seguito dall'identificatore `ao`. Al fine di risolvere questa ambiguità, decidiamo di preferire la prima interpretazione, ovvero di inglobare quanti più caratteri possibile a un token, prima di passare al prossimo. Conseguentemente, `ciao` è un singolo token identificatore. Questa scelta si chiama *coincidenza più lunga* o *longest match*.

Priorità delle regole. Non deve accadere che una stringa abbia due o più espressioni regolari che possano denotarla. In tal caso, infatti, non è chiaro quale delle due espressioni regolari, ovvero token, debba essere associata alla stringa. Si può richiedere che le espressioni regolari nella nostra enumerazione dei token denotino insiemi disgiunti. Questa richiesta è però irrealistica, perché comporta l'uso di espressioni regolari molto complesse. Per esempio, l'espressione `ID` per gli identificatori denota anche delle parole chiave del linguaggio, come `then`, che noi vorremmo invece denotare con un'espressione regolare, o token, specifica. Definire un'espressione regolare alternativa a `ID` che denoti tutti gli identificatori che non siano parole chiave è possibile ma molto complicato. È molto più semplice, invece, dire che, nella nostra enumerazione delle espressioni regolari, quelle che figurano prima hanno priorità su quelle che figurano dopo. A questo punto, è sufficiente inserire le espressioni regolari per le parole chiave prima di `ID`, per dare alle prime priorità su `ID`.

Vediamo adesso come automatizzare la costruzione di un analizzatore lessicale a partire da una specifica dei token del linguaggio data come un'enumerazione di espressioni regolari.

```
public class Symbol {
  public int sym;       // il codice che identifica il token
  public int left;      // il carattere a cui inizia
  public int right;     // il carattere a cui finisce
  public Object value;  // il valore lessicale associato, se esiste
  ...
}
```

Figura 2.2: La classe `java_cup.runtime.Symbol.java`, che rappresenta un token Kitten.

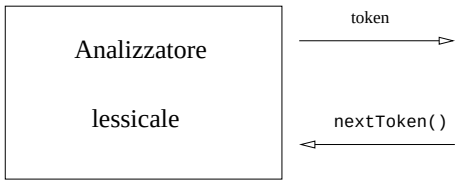


Figura 2.3: L'interfaccia dell'analizzatore lessicale.

2.3 La generazione dell'analizzatore lessicale

Per rappresentare un token, usiamo la classe in Figura 2.2, che fa parte del programma `JavaCup` che è un generatore di analizzatori sintattici. Il motivo per cui usiamo tale classe è che così facendo sarà più semplice interfacciare il nostro analizzatore lessicale con l'analizzatore sintattico che costruiremo nel Capitolo 3, dal momento che entrambi utilizzano la stessa struttura dati per rappresentare i token. I codici `sym` dei token sono enumerati dentro la classe `syntactical/sym.java`. Anche tale file fa parte del generatore di analizzatori sintattici in modo che sia l'analizzatore lessicale che quello sintattico usano gli stessi codici per i token (si veda anche la Sezione 3.2). La Figura 2.2 mostra che di ogni token è possibile conoscere la posizione `left` a cui inizia, espressa in numero di caratteri dall'inizio del file, commenti inclusi, la posizione `right` a cui finisce e l'eventuale valore lessicale `value`, se è definito per quel tipo di token. Per esempio, il token `ID(Led)` in Figura 2.1 ha `left` pari a 6, `right` pari ad 8 e `value` legato alla stringa `Led`. I token che non hanno valore lessicale avranno `value` pari a `null`.

Quello che vogliamo ottenere è un analizzatore lessicale per i token Kitten. Al posto di generare tutta la sequenza di token, come quella in Figura 2.1, e poi passarla all'analizzatore sintattico, è molto più economico generare un token alla volta e passarlo all'analizzatore sintattico. Anche quest'ultimo però dovrà essere capace di lavorare con un token alla volta. L'interfaccia dell'analizzatore lessicale sarà quindi quella mostrata in Figura 2.3. Il metodo `nextToken()` restituisce un token alla volta.

Genereremo l'analizzatore lessicale per Kitten in modo automatico, a partire dalla specifica dei token che deve riconoscere. A tal fine useremo un programma Java di nome JLex. La Figura 2.4 mostra il modo in cui generiamo l'analizzatore lessicale usando JLex. Dentro al file `lexical/Kitten.lex` enumeriamo le espressioni regolari che denotano i token del linguaggio

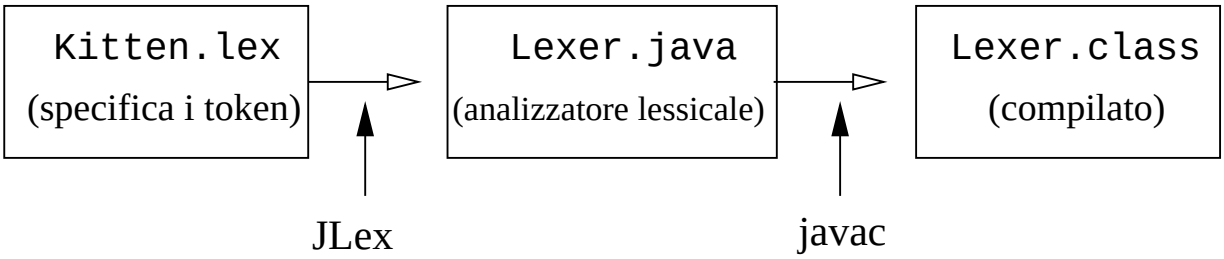


Figura 2.4: La generazione dell'analizzatore lessicale per Kitten.

Kitten, in una sintassi comprensibile dal programma JLex. Per ogni espressione regolare va fornito, nel file `lexical/Kitten.lex`, un pezzo di codice Java che viene eseguito quando viene riconosciuto il token corrispondente. Normalmente tale codice Java non fa altro che sintetizzare il token opportuno (cioè un oggetto della classe `java_cup.runtime.Symbol` in Figura 2.2) e restituirlo.

Fornendo al programma JLex la specifica `lexical/Kitten.lex` dei token, otteniamo un programma Java di nome `lexical/Lexer.java` che può essere compilato come un qualsiasi programma Java. Tale programma è l'analizzatore lessicale. Il nome `nextToken` della funzione che vogliamo generare (Figura 2.3) viene specificato scrivendo la sua specifica dentro `lexical/Kitten.lex`:

```
%function nextToken
%type java_cup.runtime.Symbol
```

con `%type` abbiamo specificato il suo tipo di ritorno.

Programmi generati in maniera automatica, come `lexical/Lexer.java`, sono normalmente di difficile lettura per un essere umano. Fidiamoci quindi del risultato, e descriviamo invece in più dettagli il contenuto del file `lexical/Kitten.lex`.

2.4 La specifica dei token

Abbiamo detto che il file `lexical/Kitten.lex` contiene la descrizione dei token Kitten, sotto forma di espressioni regolari con associata un'azione di sintesi del token corrispondente. Per esempio esso contiene le seguenti coppie espressione regolare/azione:

```
<YYINITIAL>while        {return tok(sym.WHILE, null);}
<YYINITIAL>for          {return tok(sym.FOR, null);}
....
<YYINITIAL>"+"          {return tok(sym.PLUS, null);}
<YYINITIAL>"-"          {return tok(sym.MINUS, null);}
<YYINITIAL>"*"          {return tok(sym.TIMES, null);}
<YYINITIAL>"/"          {return tok(sym.DIVIDE, null);}
```

```
  <YYINITIAL>"="          {return tok(sym.EQ, null);}
  <YYINITIAL>"!="         {return tok(sym.NEQ, null);}
  <YYINITIAL>"<"          {return tok(sym.LT, null);}
  <YYINITIAL>"<="         {return tok(sym.LE, null);}
  ....
  <YYINITIAL>":="         {return tok(sym.ASSIGN, null);}
  ....
```

che riconoscono rispettivamente le parole chiave `while`, `for`, il segno di addizione ecc. La sintassi `while` è un'espressione regolare che va intesa come `w` · `h` · `i` · `l` · `e`, cioè come la concatenazione sequenziale di cinque caratteri. La notazione `<YYINITIAL>` specifica che queste regole sono attive quando l'analizzatore lessicale è nella *modalità* di default `YYINITIAL`. Parleremo più tardi delle modalità (Sezione 2.9). Per adesso ci basta sapere che, all'inizio, l'analizzatore lessicale è in modalità `YYINITIAL`, per cui le regole precedenti sono attive. L'azione corrispondente a ciascuna regola, che viene eseguita quando il token corrispondente è stato riconosciuto, è fra parentesi graffe. Si tratta di codice Java. Per adesso, esso sintetizza il token corrispondente alle espressioni regolari, usando l'identificatore numerico unico di ciascun token (per esempio, `sym.WHILE`) e `null` come valore lessicale. Il metodo `tok` non fa altro che costruire un oggetto della classe in Figura 2.2:

```
private java_cup.runtime.Symbol tok(int kind, Object value) {
  return new java_cup.runtime.Symbol
    (kind,yychar,yychar + yylength(),value);
}
```

La variabile `yychar` contiene il numero di caratteri tra l'inizio del file e l'inizio del token. Il metodo `yylength()` ritorna la lunghezza del token riconosciuto. Metodi di ausilio, come quello precedente, sono inseriti in `lexical/Kitten.lex` fra i delimitatori `%{` e `}%` e vengono ricopiati testualmente da JLex dentro `lexical/Lexer.java`.

I token che hanno un valore lessicale sono specificati in maniera appena più complicata:

```
<YYINITIAL>[a-zA-Z][a-zA-Z0-9_]*
    {return tok(sym.ID,yytext());}
<YYINITIAL>[0-9]+
    {return tok(sym.INTEGER,new Integer(yytext()));}
<YYINITIAL>[0-9]*"."[0-9]+
    {return tok(sym.FLOATING,new Float(yytext()));}
```

Si noti come il valore lessicale degli identificatori `ID` sia la stringa che rappresenta l'identificatore, mentre per interi e numeri a virgola mobile si tratta, rispettivamente, di un oggetto di classe `java.lang.Integer` e `java.lang.Float`. Il programma JLex accorda maggiore priorità alle regole specificate prima in `lexical/Kitten.lex`. Al fine, per esempio, di non fare riconoscere la parola chiave `while` come un identificatore, occorre mettere la regola per il `while` prima di quella per gli identificatori (Sezione 2.2).

Esiste infine una regola che riconosce qualsiasi carattere ma che, essendo messa alla fine, viene eseguita solo quando nessun'altra regola è applicabile. Tale regola segnala un'errore lessicale, cioè la lettura di un carattere che non è associabile ad alcun token:

```
<YYINITIAL>.            {errorMsg.error(yychar,"Unmatched input");}
```

Il carattere `.` (punto) è un'espressione regolare che denota l'insieme dei caratteri dell'alfabeto. La si può immaginare come un'abbreviazione dell'alternanza fra tutti i caratteri dell'alfabeto.

Se si dovessero aggiungere nuovi token all'enumerazione contenuta nel file `lexical/Kitten.lex`, occorre fare attenzione alla posizione in cui le loro espressioni regolari vengono inserite. Molti studenti tendono a inserire queste nuove espressioni regolari in fondo, dopo la regola che usa il carattere punto. Questa è la peggior scelta che si può fare: se il token è formato da un unico carattere, esso non verrà mai riconosciuto perché la regola col punto avrà priorità sulla nuova regola (Sezione 2.2). Se il token è formato da più caratteri alfabetici, esso non verrà mai riconosciuto perché la regola per l'identificatore avrà priorità sulla nuova regola. È quindi consigliabile inserire le espressioni regolari per nuovi token subito dopo l'enumerazione della punteggiatura, prima degli identificatori.

2.5 La segnalazione di errori

Abbiamo appena visto che l'analizzatore lessicale può avere bisogno di segnalare un errore all'utente di Kitten. Lo stesso (e molto più spesso) accadrà con l'analizzatore sintattico e con quello semantico. Tutti questi analizzatori usano la stessa classe `errorMsg/ErrorMsg.java` per segnalare errori. La sua interfaccia[1] è in Figura 2.5. Il metodo `error()` segnala un errore all'utente. La posizione dell'errore è indicata all'utente con la notazione `riga:colonna`. Ma il metodo `error()` richiede solo il numero `pos` di caratteri passati dall'inizio del file che si sta compilando. Per potere trasformare `pos` in `riga:colonna`, occorre che l'oggetto di segnalazione di errori sia al corrente di dove, nel file sorgente, si trovano i caratteri di newline. Ecco perché, ogni volta che si incontra tale carattere, l'analizzatore lessicale chiama il metodo `newline()`:

```
<YYINITIAL>\n           {errorMsg.newline(yychar);}
```

Si noti che questa regola ha anche l'effetto secondario di scartare il carattere newline, poiché non vogliamo i caratteri di spaziatura nel risultato dell'analisi lessicale (Figura 2.1). Conoscendo le posizioni dei caratteri di newline, è possibile sapere quanti newline occorrono nei primi `pos` caratteri del file sorgente ed è quindi possibile recuperare l'informazione di `riga`. La `colonna` sarà il numero di caratteri tra l'ultimo newline e `pos`.

Il campo `errorMsg` dell'analizzatore lessicale contiene la sua struttura di segnalazione di errore. Essa è creata dal costruttore di quest'ultimo a partire dal nome del file che si sta compilando.

[1] Per più informazioni sulle classi e i metodi di Kitten, ricordiamo che è disponibile la documentazione JavaDoc dentro la directory `javadocs` della distribuzione di Kitten.

```
public class ErrorMsg {
  /* costruttore: si chiede il nome del file che si sta compilando */
  public ErrorMsg(String fileName) { ... }

  /* chiamata quando si incontra un newline in fileName */
  public newline(int pos) { ... }

  /* segnala un errore msg alla posizione pos dall'inizio di fileName */
  public error(int pos, String msg) { ... }

  /* dice se si e' verificato almeno un errore */
  public static boolean anyErrors() { ... }
}
```

Figura 2.5: La classe `errorMsg.ErrorMsg.java` per la segnalazione di errori.

2.6 JLex: da espressioni regolari ad automi finiti non deterministici

Abbiamo visto che JLex trasforma una sequenza di espressioni regolari in un programma Java (l'analizzatore lessicale) capace di riconoscere i token denotati da tali espressioni regolari. Vediamo adesso di capire come funziona questo programma.

Le espressioni regolari sono degli ottimi strumenti per *descrivere* un insieme di stringhe, il loro linguaggio, ma certamente non per *riconoscere* tale insieme: data una stringa, vogliamo sapere se appartiene o meno al linguaggio generato da una data espressione regolare. Al fine di riconoscere un linguaggio, useremo degli *automi a stati finiti*, che ammettono anche una semplice implementazione tramite un calcolatore.

Definizione 6 (Automa Finito non Deterministico)**.** Un *automa finito non deterministico* su un alfabeto Λ è un grafo orientato finito i cui nodi sono detti *stati* e i cui archi, detti *transizioni*, sono etichettati con un carattere in Λ o con ε. Un nodo del grafo è identificato come *iniziale* e un insieme di nodi del grafo come *finali*.

La Figura 2.6 mostra un automa finito non deterministico sul linguaggio $\Lambda = \{\mathsf{a}, \mathsf{b}\}$. Il nodo iniziale è individuato da una freccia entrante. I nodi finali sono individuati con una doppia cerchiatura.

Un *percorso* in un automa è una sequenza di nodi legati da archi.

Definizione 7 (Percorso)**.** Un *percorso* in un automa finito non deterministico è una sequenza di nodi $n_1 \rightarrow^{c_1} n_2 \rightarrow^{c_2} \cdots \rightarrow^{c_{k-1}} n_k$ tale che per ogni $i = 1, \ldots, k-1$ esiste un arco $n_i \rightarrow^{c_i} n_{i+1}$ fra n_i ed n_{i+1}. La *stringa espressa* da un percorso è la concatenazione delle etichette sugli archi che passano per i nodi del percorso, cioè $c_1 c_2 \cdots c_{k-1}$.

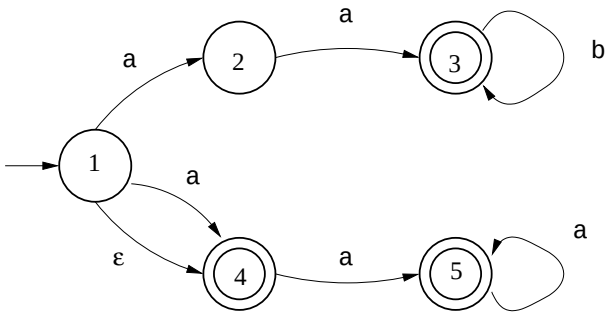


Figura 2.6: Un automa finito non deterministico.

Per esempio, l'automa in Figura 2.6 possiede un percorso $4 \rightarrow^{\mathtt{a}} 5 \rightarrow^{\mathtt{a}} 5 \rightarrow^{\mathtt{a}} 5$ che esprime la stringa `aaa`. Possiamo quindi definire il linguaggio *accettato* da un automa come l'insieme delle stringhe espresse da un percorso dell'automa che comincia nel suo unico nodo iniziale e termina in un nodo finale.

Definizione 8 (Linguaggio Accettato da un Automa)**.** Il linguaggio $\mathcal{L}(A)$ *accettato* da un automa non deterministico A è

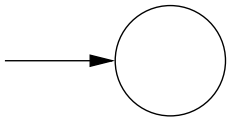
$$\mathcal{L}(A) = \left\{ s \in \Lambda^* \;\middle|\; \begin{array}{l} s = c_1 \cdots c_{k-1} \text{ ed esiste un percorso } n_1 \rightarrow^{c_1} \cdots \rightarrow^{c_{k-1}} n_k \text{ di } A \\ \text{tale che } n_1 \text{ è il nodo iniziale di } A \text{ ed } n_k \text{ è un nodo finale di } A \end{array} \right\}.$$

Per esempio, possiamo determinare il linguaggio accettato dall'automa in Figura 2.6 considerando l'unione dei linguaggi accettati in ciascuno dei suoi tre stati di accettazione. Essa è il linguaggio fatto dalle stringhe che cominciano con due `a` e continuano con un numero arbitrario (anche nullo) di `b`, dalle stringhe che cominciano con una o due `a` e continuano con un numero arbitrario (anche nullo) di `a`, e dalla stringa vuota. Si noti che un automa può accettare una stringa tramite vari percorsi differenti. Per esempio, l'automa in Figura 2.6 accetta la stringa `a` tramite il percorso $1 \rightarrow^{\mathtt{a}} 4$ ma anche tramite il percorso $1 \rightarrow^{\varepsilon} 4 \rightarrow^{\mathtt{a}} 5$.

Il linguaggio accettato dall'automa in Figura 2.6 è in effetti quello denotato dall'espressione regolare $\varepsilon|\mathtt{aab}^*|\mathtt{aa}^*|\mathtt{aaa}^*$ (che sarebbe possibile semplificare in $\mathtt{aab}^*|\mathtt{a}^*$). Questa non è una coincidenza: si può in effetti dimostrare che, dato un linguaggio, esiste un automa finito non deterministico che lo accetta se e solo se esiste un'espressione regolare che lo denota. Di questo risultato vediamo adesso solo come è possibile costruire un automa finito non deterministico a partire da una espressione regolare, in modo che quest'ultima denoti lo stesso linguaggio accettato dall'automa. Più in dettaglio, forniamo una definizione induttiva di un automa finito non deterministico *indotto* da una data espressione regolare.

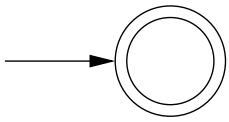
Procediamo per induzione sulla struttura delle espressioni regolari, definendo un automa non deterministico corrispondente a ciascun tipo di espressione regolare della Definizione 2. In questa costruzione manterremo l'invariante che l'automa costruito avrà sempre *al più uno stato di accettazione*.

L'espressione regolare $\emptyset$ denota il linguaggio vuoto (Definizione 4). Un automa che accetta lo stesso linguaggio è il seguente:



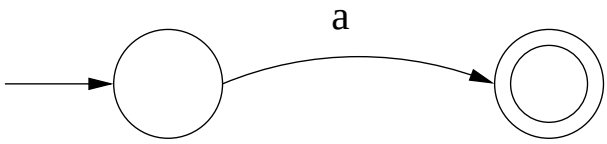
Esso non ha stati di accettazione e conseguentemente accetta l'insieme vuoto di stringhe.

L'espressione regolare ε denota un linguaggio che contiene la sola stringa ε. Esso è lo stesso linguaggio accettato dall'automa

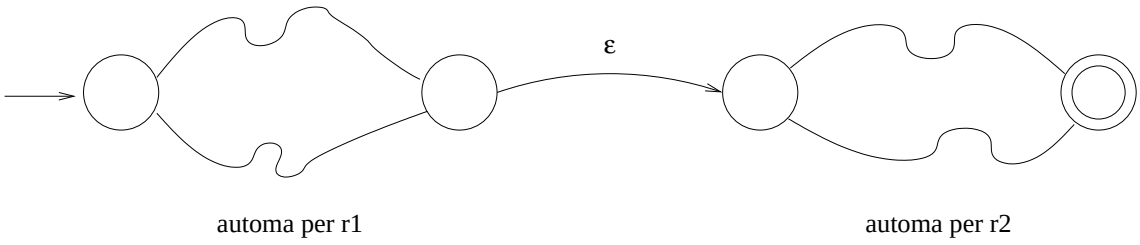


Si noti che lo stato iniziale e quello di accettazione di questo automa coincidono.

L'espressione regolare a, con $a \in \Lambda$, denota un linguaggio formato dalla sola stringa a. Esso è lo stesso linguaggio accettato dall'automa

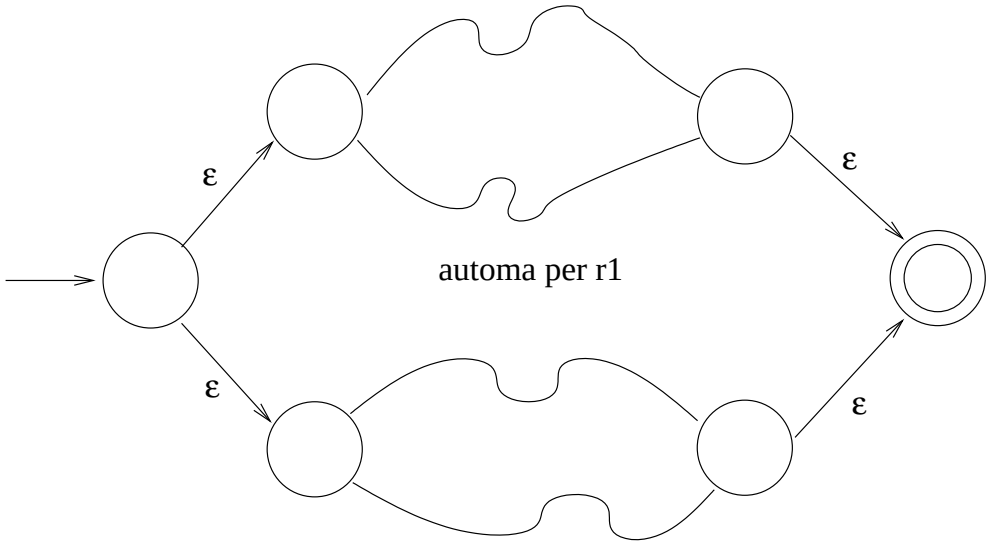


L'espressione regolare $r_1 r_2$, cioè la sequenza di due espressioni regolari r_1 ed r_2, denota il linguaggio formato dalle stringhe ottenute concatenando una stringa del linguaggio denotato da r_1 con una stringa del linguaggio denotato da r_2. Otteniamo quindi un automa che accetta lo stesso linguaggio concatenando sequenzialmente l'automa corrispondente ad r_1 con l'automa corrispondente ad r_2:



Si noti che lo stato di accettazione dell'automa corrispondente ad r_1 non è più di accettazione nell'automa composto per $r_1 r_2$. Se inoltre r_1 non ha stati di accettazione, allora la transizione etichettata con ε non viene aggiunta.

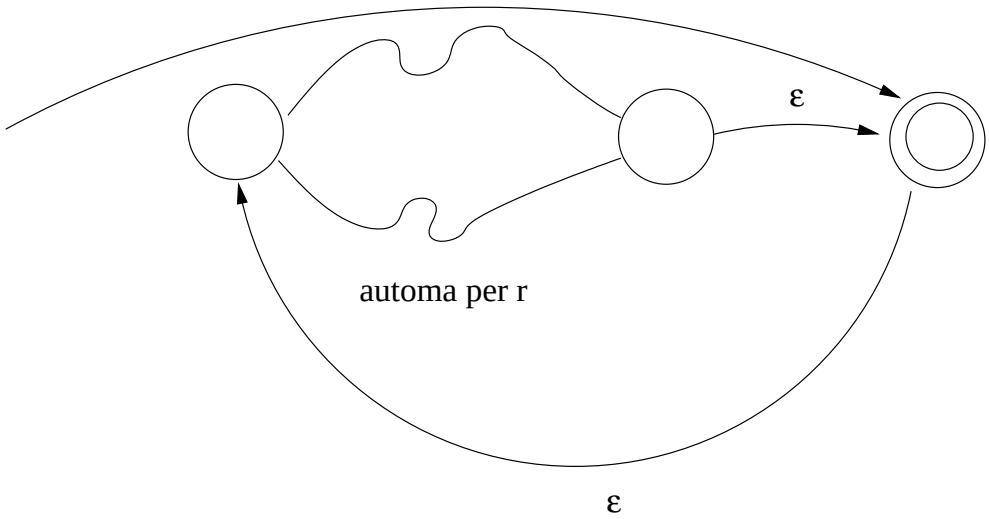
L'espressione regolare $r_1|r_2$ denota l'unione dei linguaggi di r_1 e di r_2. Otteniamo quindi un automa che accetta lo stesso linguaggio mettendo in alternativa gli automi corrispondenti alle espressioni regolari r_1 ed r_2:



automa per r2

Si noti che gli stati di accettazione degli automi corrispondenti ad r_1 ed r_2 non sono più di accettazione nell'automa composto per $r_1|r_2$, mentre un nuovo stato di accettazione è stato aggiunto in quest'ultimo. Se inoltre gli automi per r_1 o r_2 non hanno stati di accettazione allora non si aggiunge la freccia (o le frecce) etichettate con ε che portano nello stato di accettazione.

L'espressione regolare r^* denota il linguaggio ottenuto ripetendo un numero arbitrario di volte le stringhe del linguaggio denotato da r. Otteniamo un automa che accetta lo stesso linguaggio creando un ciclo sull'automa corrispondente ad r. Questo ciclo può essere percorso un numero arbitrario di volte, eventualmente anche nessuna volta:



Si noti che lo stato di accettazione dell'automa per r non è più di accettazione nell'automa per r^*, e che in quest'ultimo lo stato finale e quello iniziale coincidono. Se inoltre l'automa corrispondente ad r non avesse alcuno stato di accettazione, non si mette la transizione etichettata con ε che porta nello stato di accettazione.

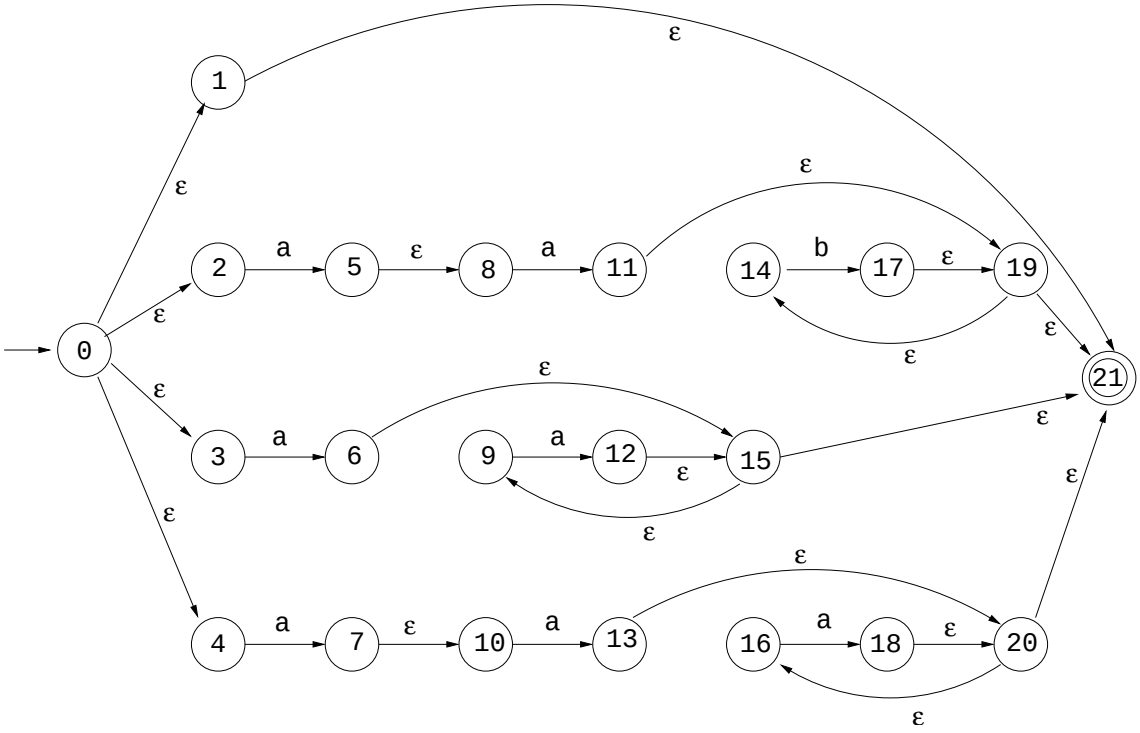


Figura 2.7: Un automa finito non deterministico costruito induttivamente a partire dall'espressione regolare $\varepsilon|\texttt{aab}^*|\texttt{aa}^*|\texttt{aaa}^*$. La numerazione dei nodi è arbitraria.

Si consideri per esempio l'espressione regolare $\varepsilon|\texttt{aab}^*|\texttt{aa}^*|\texttt{aaa}^*$. Abbiamo già notato che essa denota lo stesso linguaggio riconosciuto dall'automa in Figura 2.6. Il risultato della costruzione esplicita di un automa corrispondente a tale espressione regolare, usando le regole induttive di costruzione che abbiamo appena descritto, è mostrato in Figura 2.7. L'automa in Figura 2.7 è diverso da quello in Figura 2.6. In particolare, esso contiene più stati e transizioni. I due automi sono però *equivalenti*, nel senso che essi accettano lo stesso linguaggio (Definizione 8).

2.7 JLex: da automi finiti non deterministici ad automi finiti deterministici

La nozione di *automa* che abbiamo dato nella Definizione 6 caratterizza automi finiti *non deterministici* in quanto è possibile che ci siano più transizioni uscenti da uno stesso stato etichettate con lo stesso carattere dell'alfabeto o transizioni etichettate con ε. Se quindi tali automi sono utili per *descrivere* un linguaggio, essi sono però scomodi per *riconoscere* un linguaggio, cioè per fornire una procedura effettiva che permetta di determinare se una stringa appartiene o meno al linguaggio che essi denotano. Occorrerebbe infatti a tal fine considerare tutti i percorsi possibili nell'automa.

Se limitassimo ad al più uno il numero delle transizioni uscenti da uno stesso stato etichettate con un dato carattere e vietassimo delle transizioni etichettate con ε, otterremmo un tipo di automa che ci permette di riconoscere se una stringa appartiene al linguaggio che esso denota semplicemente cercando l'*unico* percorso nell'automa per tale stringa, se esiste. Questo nuovo tipo di automa è detto *deterministico*.

Definizione 9 (Automa Finito Deterministico)**.** Un *automa finito deterministico* è un automa finito non deterministico senza transizioni etichettate con ε e tale che per ciascun nodo e ciascun carattere dell'alfabeto c'è al più una transizione uscente dal nodo etichettata con tale carattere.

Per esempio, l'automa in Figura 2.6 sarebbe deterministico se non ci fosse la transizione etichettata con ε e se le due transizioni uscenti dal nodo 1 ed etichettate con `a` avessero etichette diverse.

Dal momento che un automa finito deterministico è un caso particolare di automa finito non deterministico, le definizioni di *percorso* (Definizione 7) e di *linguaggio accettato* (Definizione 8) sono valide anche per gli automi finiti deterministici. È quindi immediato osservare che se un linguaggio è accettato da un automa deterministico allora esso è accettato anche da un automa non deterministico (lo stesso automa!). Il viceversa è anch'esso vero, benché meno immediato da dimostrare e intuitivamente meno ovvio. In effetti è possibile simulare un automa finito non deterministico con un automa deterministico i cui stati *rappresentano* insiemi di stati dell'automa non deterministico simulato. Questo significa che il non determinismo non aggiunge potenza espressiva agli automi a stati finiti.

La trasformazione di un automa non deterministico in uno deterministico comporta la definizione della *ε-chiusura* e della *transizione su un carattere* di un insieme di stati.

Definizione 10 (ε-chiusura e transizione su un carattere)**.** Dato un automa finito non deterministico A e un insieme N dei suoi nodi, la *ε-chiusura* $\varepsilon(N)$ di N è l'insieme N stesso unito all'insieme dei nodi raggiungibili da un nodo di N usando solo transizioni etichettate con ε:

$$\varepsilon(N) = \{n' \in A \mid \text{esiste } n \in N \text{ tale che } n \to^{\varepsilon} \cdots \to^{\varepsilon} n'\}.$$

Se $c \in \Lambda$, la *transizione su c di N*, indicata come $c(N)$, è l'insieme dei nodi di A raggiungibili da un nodo di N tramite una transizione etichettata con c seguita da un numero arbitrario di transizioni etichettate con ε:

$$c(N) = \varepsilon(\{n' \mid \text{esiste } n \in N \text{ tale che } n \to^{c} n'\}).$$

Per esempio, in Figura 2.7 abbiamo

$$\begin{aligned}
\varepsilon(\{0\}) &= \{0, 1, 2, 3, 4, 21\} \\
\varepsilon(\{5, 8\}) &= \{5, 8\} \\
\mathtt{a}(\{0, 1, 2, 3, 4, 21\}) &= \{5, 6, 7, 8, 9, 10, 15, 21\} \\
\mathtt{b}(\{0, 1, 2, 3, 4, 21\}) &= \emptyset.
\end{aligned}$$

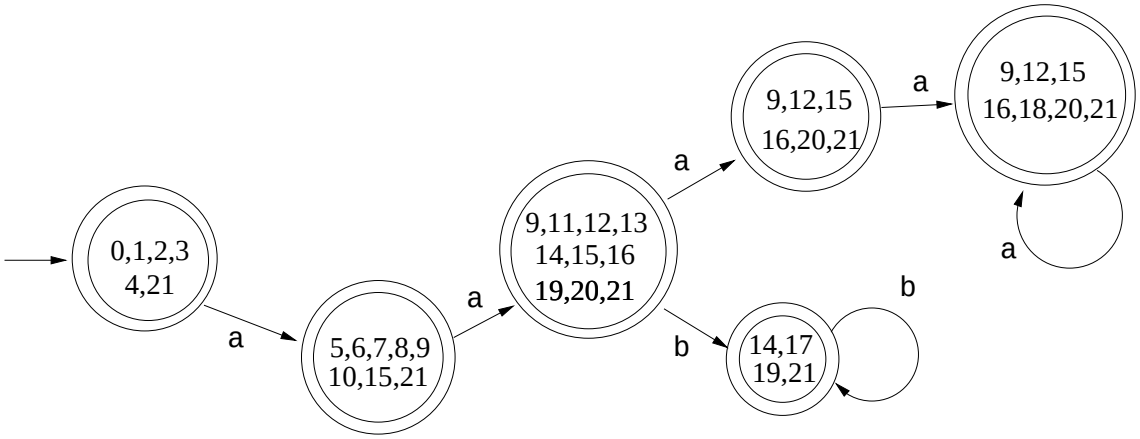


Figura 2.8: Un automa finito deterministico ottenuto dall'automa non deterministico in Figura 2.7.

È possibile trasformare un automa finito non deterministico A in uno deterministico A' equivalente i cui stati sono insiemi di stati di A. Lo stato iniziale di A' e $\varepsilon(i)$, dove i è lo stato iniziale di A. Da ogni stato s di A' e per ogni carettere c dell'alfabeto, esce da s una transizione etichettata con c che porta nello stato $c(s)$ di A'. Se $c(s) = \emptyset$ allora si può fare a meno di indicare la transizione. Gli stati di accettazione di A' sono quelli che contengono almeno uno degli stati di accettazione di A. Per esempio, l'automa finito non deterministico in Figura 2.7 viene trasformato in quello deterministico mostrato in Figura 2.8. Si noti che tutti i suoi stati sono di accettazione.

La trasformazione da espressione regolare ad automa della Sezione 2.6 fornisce in genere un automa finito *non deterministico* a causa delle regole di traduzione dell'alternanza di due espressioni regolari. Tale automa può poi essere trasformato in un automa finito deterministico equivalente con la trasformazione appena descritta. Questo è proprio quello che fa JLex, ottenendo un automa deterministico che può essere più comodamente eseguito su una macchina sequenziale. Va detto comunque che, per una maggiore efficienza, JLex evita di costruire l'automa non deterministico ma costruisce invece direttamente quello deterministico. Si tratta comunque di una ottimizzazione della trasformazione fin qui descritta.

In genere l'automa ottenuto eliminando il non determinismo potrebbe essere *ridondante*, nel senso che esso può contenere due stati con identiche funzioni, che possono quindi venire fusi in un unico stato. Questo è per esempio il caso degli stati $\{9, 12, 15, 16, 20, 21\}$ e $\{9, 12, 15, 16, 18, 20, 21\}$ in Figura 2.8. Essi potrebbero essere fusi in un unico stato con una transizione uscente per `a` che porta sullo stato stesso, riducendo il numero di stati dell'automa. L'*ottimizzazione* del numero di stati dell'automa deterministico risultante dalla trasformazione viene quindi effettuata da JLex al fine di ridurre la dimensione dell'automa deterministico finale. Quest'ultimo viene infine scritto nel file `Lexer.java` (Figura 2.4) sotto forma di un sorgente Java che contiene l'insieme dei nodi e la tabella di transizione dell'automa.

Occorre adesso comprendere l'ultimo aspetto del funzionamento di JLex. Esso infatti per-

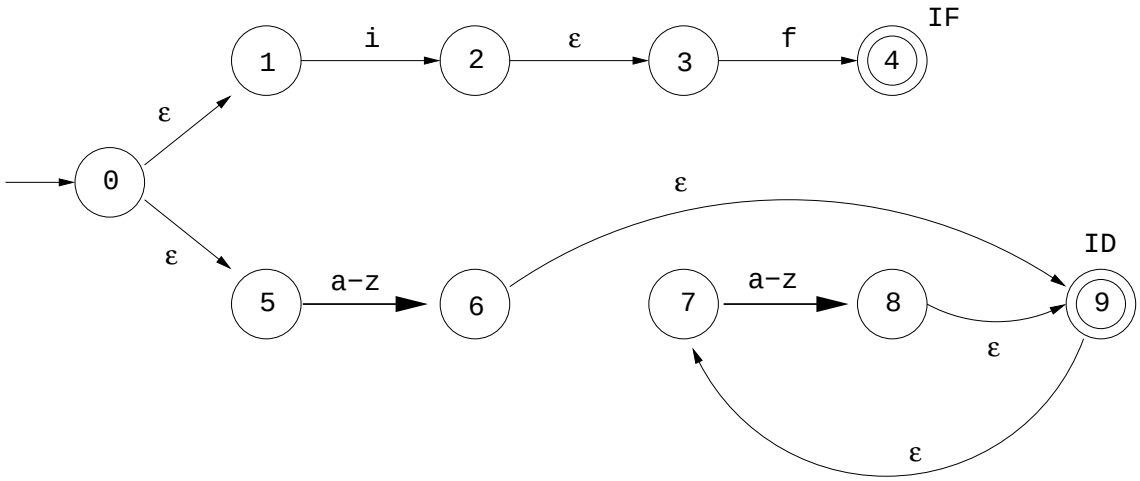


Figura 2.9: Un automa finito non deterministico per i token `IF` ed `ID`.

mette di riconoscere un *insieme* di token e non una sola espressione regolare. Inoltre esso implementa, fra tali token, i meccanismi di coincidenza più lunga e priorità delle regole descritti nella Sezione 2.2. Consideriamo questi aspetti nella sezione seguente.

2.8 JLex: la costruzione di un automa non deterministico per un insieme di token

In Sezione 2.6 abbiamo visto come un'espressione regolare può essere trasformata in un automa finito non deterministico che accetta lo stesso linguaggio denotato dall'espressione regolare. In Sezione 2.7 abbiamo visto poi come tale automa possa venire trasformato in un automa equivalente ma deterministico e quindi di più facile implementazione su una macchina sequenziale. In questa sezione vediamo come JLex applica queste tecniche per riconoscere un insieme di token.

JLex costruisce l'automa finito non deterministico corrispondente all'espressione regolare che denota ciascun token, usando la tecnica descritta nella Sezione 2.6. Tali automi vengono poi messi in alternanza creando un unico, grande automa avente uno stato iniziale legato agli stati iniziali di ciascun automa. Per esempio, per i due token

```
IF      if
ID      [a-z][a-z]*
```

il programma JLex costruisce l'automa in Figura 2.9. Abbiamo etichettato alcune transizioni con un intervallo di caratteri, il cui senso è che esse rappresentano in effetti un *insieme* di transizioni, una per ciascun carattere nell'intervallo. Si noti inoltre che abbiamo annotato, accanto a ciascuno stato di accettazione, qual è il token accettato in tale stato. JLex trasforma quindi tale automa in un automa finito deterministico, con la tecnica descritta nella Sezione 2.7. Il risultato, per

il nostro esempio, è mostrato in Figura 2.10. È possibile che uno stesso stato dell'automa deterministico venga etichettato con più token, poiché contiene lo stato finale di più sotto-automi. Questo è il caso dello stato etichettato con `IF` ed `ID` in Figura 2.10. Questo significa che se si arriva in tale stato allora i caratteri letti dall'inizio del file possono essere considerati sia come un'istanza del token `ID` che come un'istanza del token `IF`. Abbiamo già visto come si risolve questa ambiguità: dando precedenza al token che figura prima nell'enumerazione (Sezione 2.2). Nel nostro caso, il token `IF` è stato enumerato per prima e quindi l'etichetta `ID` viene rimossa e in tale stato si riconosce solo il token `IF`.

Il risultato dell'elaborazione effettuata da JLex è quindi un automa come quello in Figura 2.10, scritto in Java. Ogni volta che si chiama il metodo `nextToken()`, tale automa viene eseguito a partire dalla posizione corrente nel file sorgente. Quando occorre fermarsi in questa lettura? Ci si potrebbe fermare non appena si finisce in uno stato di accettazione. Ma questo significherebbe che `abc` verrebbe riconosciuto come tre token `ID` piuttosto che come un unico token `ID`. JLex decide quindi di avanzare nel file di input finché esiste una transizione possibile nell'automa. Quando non esiste più alcuna transizione possibile, si restituisce il token che etichettava l'ultimo stato di accettazione per cui si è passati. Questo modo di procedere implementa quindi la *coincidenza più lunga* della Sezione 2.2. Se nessuno stato di accettazione è stato ancora incontrato, JLex dà un messaggio di errore. Si noti comunque che quest'ultima situazione è comunque impossibile nel caso di Kitten poiché nella enumerazione dei token abbiamo inserito una regola di default che accetta qualsiasi carattere (Sezione 2.4).

L'implementazione dell'automa da parte di JLex deve tener conto di un ultimo problema: i caratteri letti dopo l'ultimo stato di accettazione per cui si è passati vanno *rimessi* nel file di input per essere processati alla prossima richiesta di `nextToken()`. A tal fine basta utilizzare due puntatori nel file di input: uno che punta all'ultimo stato di accettazione per cui si è passati e uno che punta all'ultimo carattere letto. Prima di terminare una chiamata a `nextToken()`, JLex ha cura di riportare l'ultimo puntatore a coincidere col primo.

2.9 Modalità lessicali: commenti e stringhe

Le regole contenute in `lexical/Kitten.lex` hanno come prefisso una *modalità* che indica quando esse sono attive. Normalmente, l'analizzatore lessicale è nella modalità di default `YYINITIAL`. È possibile però cambiare modalità tramite il metodo `yybegin()`. Occorre per prima cosa dichiarare le nuove modalità dentro `lexical/Kitten.lex`:

```
%state COMMENT
%state STRING
```

La scelta di queste due modalità è finalizzata a semplificare la gestione di commenti e stringhe. Per esempio, la modalità `COMMENT` si attiva quando incontriamo la sequenza di caratteri `/*`:

```
<YYINITIAL>"/*"          {commentCount++; yybegin(COMMENT);}
```

La variabile `commentCount` conta il livello di annidamento dei commenti visti fino a questo momento. Essa è dichiarata fra i delimitatori `%{` e `%}` e inizializzata a 0.

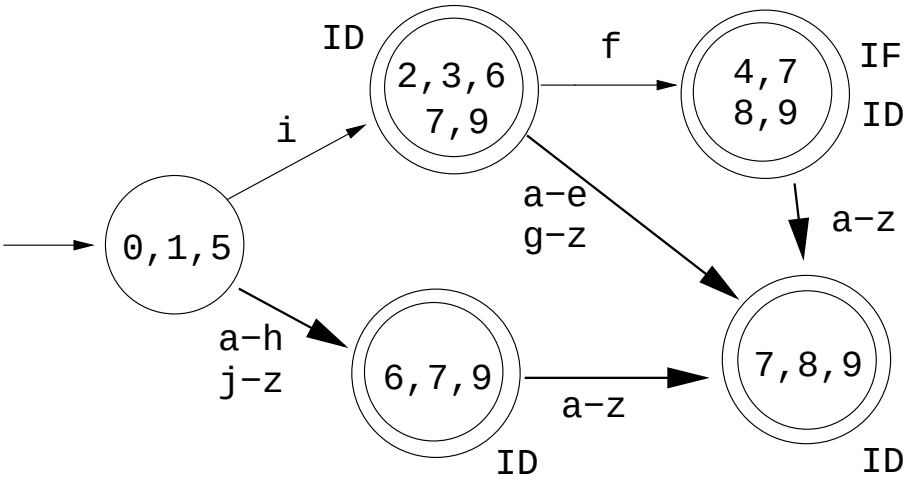


Figura 2.10: L'automa finito deterministico costruito a partire dall'automa non deterministico in Figura 2.9.

Le uniche regole attive in modalità `COMMENT` hanno come scopo di scartare tutti i caratteri letti fino alla chiusura dell'ultimo commento, tenendo conto dell'annidamento. Non dobbiamo però dimenticare di registrare le posizioni dei caratteri di newline:

```
<COMMENT>"*/"          {commentCount--;
                        if (commentCount == 0) yybegin(YYINITIAL);}
<COMMENT>"/*"          {commentCount++;}
<COMMENT>\n            {newline();}
<COMMENT>.             {}
```

Occorre evitare che il file sorgente termini con un commento ancora aperto. A tal fine usiamo il fatto che, quando l'analizzatore lessicale giunge alla fine del file sorgente, viene eseguito il codice specificato, dentro `lexical/Kitten.lex`, fra i delimitatori `%eofval{` e `%eofval}`. Nel nostro caso abbiamo scelto di eseguire quanto segue:

```
%eofval{
        {
          if (commentCount != 0) err("Unclosed comment");
          return tok(sym.EOF, null);
        }
%eofval}
```

ovvero controlliamo che il file non termini con un commento ancora aperto e poi restituiamo comunque il token fittizio `EOF`.

La modalità `STRING` si attiva quando si incontra un carattere di doppio apice. Essa riconosce la stringa fra doppi apici e processa le sequenze di escape. Memorizza il valore lessicale dentro una variabile `myString` che viene usata come valore lessicale del token `STRING`:

```
<YYINITIAL>"\"" {myString = ""; yybegin(STRING);}
<STRING>\\n     {myString += "\n";}
<STRING>\\t     {myString += "\t";}
... altre sequenze di escape ...
<STRING>"\""    {yybegin(YYINITIAL); return tok(sym.STRING,myString);}
<STRING>\n      {errorMsg.newline(yychar); myString += "\n";}
<STRING>.       {myString += yytext();}
```

La seconda e la terza regola inseriscono un carattere di escape dentro la stringa quando viene riconosciuto la corrispondente espressione di escape dentro il file sorgente. Si noti che l'espressione regolare `\\n` è formata da *due* caratteri `\\` e `n`. Il primo è a sua volta una sequenza di escape di JLex per esprimere il carattere `\`. Esistono altre espressioni di escape per inserire, per esempio, un carattere a partire dal suo codice ASCII. Esse sono esaminabili dentro `lexical/Kitten.lex`. La terz'ultima regola riporta l'analizzatore in modalità `YYINITIAL` quando si incontra il carattere doppio apice di chiusura della stringa. Le ultime due regole accumulano tutti caratteri dentro `myString`.

L'uso di comandi Java con memoria, come l'assegnamento su `commentCount` e `myString`, ha l'effetto di aumentare la potenza di JLex, al punto che si potrebbe pensare di affidargli compiti molto più complessi dell'analisi lessicale. Per esempio, la stessa analisi sintattica (Capitolo 3) potrebbe essere effettuata all'interno di JLex. In effetti, l'uso di variabili Java dà a JLex la capacità di superare il potere espressivo limitato delle espressioni regolari o degli automi a stati finiti, per accedere all'espressività superiore di sistemi di riconoscimento di linguaggi basati su una quantità di memoria potenzialmente infinita, come gli automi a pila. Va però osservato che è più semplice limitare l'uso di JLex allo scopo per cui è stato pensato, cioè all'analisi lessicale, con qualche *concessione* all'uso di variabili Java per commenti e costanti stringhe. Nel Capitolo 3 useremo uno strumento più adeguato, chiamato `JavaCup`, per descrivere e riconoscere la struttura sintattica di Kitten.

2.10 L'uso di JLex

Una volta inserita dentro `lexical/Kitten.lex` la specifica dell'analizzatore lessicale che vogliamo generare, non ci rimane che generarlo tramite JLex. A tal fine basta eseguire JLex:

```
java JLex.Main lexical/Kitten.lex
```

Per default, JLex scrive l'analizzatore lessicale nel file `lexical/Kitten.lex.java`, che contiene la dichiarazione di una classe di nome `Yylex`. Basta quindi ridenominare tale file in `lexical/Lexer.java`, dopo avere però sostituito tutte le stringhe `Yylex` al suo interno con la stringa `Lexer`. A tal fine usiamo il programma Unix `sed`:

```
cat lexical/Kitten.lex.java | sed s/Yylex/Lexer/g > lexical/Lexer.java
rm lexical/Kitten.lex.java
```

Questi tre comandi sono stati inseriti dentro a un makefile al fine di semplificarne l'uso. Basta quindi scrivere `make lexical` per ottenere lo stesso effetto.

In conclusione, abbiamo ottenuto un analizzatore lessicale `lexical/Lexer.java` che contiene un costruttore

```
public Lexer(fileName)
```

e un metodo

```
public java_cup.runtime.Symbol nextToken()
```

che estrae e restituisce un token alla volta da `fileName` (Figura 2.3).

Esercizio 5. Si consideri il linguaggio $\Lambda = \{\mathtt{0}, \mathtt{1}\}$. Si definisca un'espressione regolare che denota tutti e soli i numeri binari dispari.

Esercizio 6. In Sezione 2.6 abbiamo trasformato la sequenza $r_1 r_2$ di due espressioni regolari r_1 ed r_2 in un automa ottenuto legando gli automi ottenuti per r_1 ed r_2, rispettivamente, con una transizione etichettata con ε. Si provi a giustificare perché tale transizione è necessaria e non può invece essere eliminata fondendo lo stato finale dell'automa per r_1 con lo stato iniziale dell'automa per r_2.

Esercizio 7. In Sezione 2.6 abbiamo trasformato l'iterazione r^* di un'espressione regolare r in un automa ottenuto a partire dall'automa per r. Abbiamo però aggiunto un nuovo stato terminale e, al contempo, iniziale. Si provi a giustificare perché tale stato è necessario e non è possibile invece usare come stato iniziale e terminale lo stato finale dell'automa per r.

Esercizio 8. Si definisca, usando le tecniche descritte in questo capitolo, un automa non deterministico che accetta i token `CONST`, `CONTINUE` ed `ID`.

Esercizio 9. Si consideri il linguaggio delle cifre decimali. Si definisca un'automa finito deterministico che accetta tutti i soli i numeri decimali divisibili per 3.

Esercizio 10. Si trasformi l'automa non deterministico dell'esercizio 8 in un automa finito deterministico.

Capitolo 3

Analisi Sintattica

Abbiamo visto nel Capitolo 2 come la sequenza di caratteri di un sorgente Kitten viene trasformata in una lista di *token*. Programmi *lessicalmente errati*, cioè contenenti sequenze di caratteri che non compongono alcun token, vengono rifiutati dall'analizzatore lessicale con una segnalazione di errore.

Non bisogna però pensare che l'analizzatore lessicale impedisca al programmatore di scrivere programmi *sintatticamente errati*. Per esempio, il comando `while (a != b a := a + 1` viene tradotto nella sequenza di token `WHILE`, `LPAREN`, `ID`, `NEQ`, `ID`, `ID`, `ASSIGN`, `ID`, `PLUS`, `INTEGER` senza che alcun messaggio di errore venga segnalato dall'analizzatore lessicale. Ciò nonostante, tale comando è sintatticamente errato in quanto contiene una parentesi tonda aperta che non è stata richiusa. In questo capitolo intendiamo presentare delle tecniche che permettono di segnalare al programmatore errori sintattici come quello appena mostrato. Tali tecniche sono chiamate tecniche di analisi sintattica o di *parsing*. L'analisi sintattica ha due scopi:

- garantire che il codice rispetti le regole sintattiche del linguaggio. Se così non è, un errore di sintassi deve essere segnalato al programmatore;
- costruire una rappresentazione *strutturata* del programma, detta *sintassi astratta* del programma, che può essere *comodamente* usata dalle successive fasi di analisi semantica e di generazione e ottimizzazione del codice.

Si consideri per esempio il programma in Figura 1.4. L'analisi sintattica di Kitten accetta tale programma come sintatticamente corretto e genera la sua struttura logica mostrata in Figura 3.1.

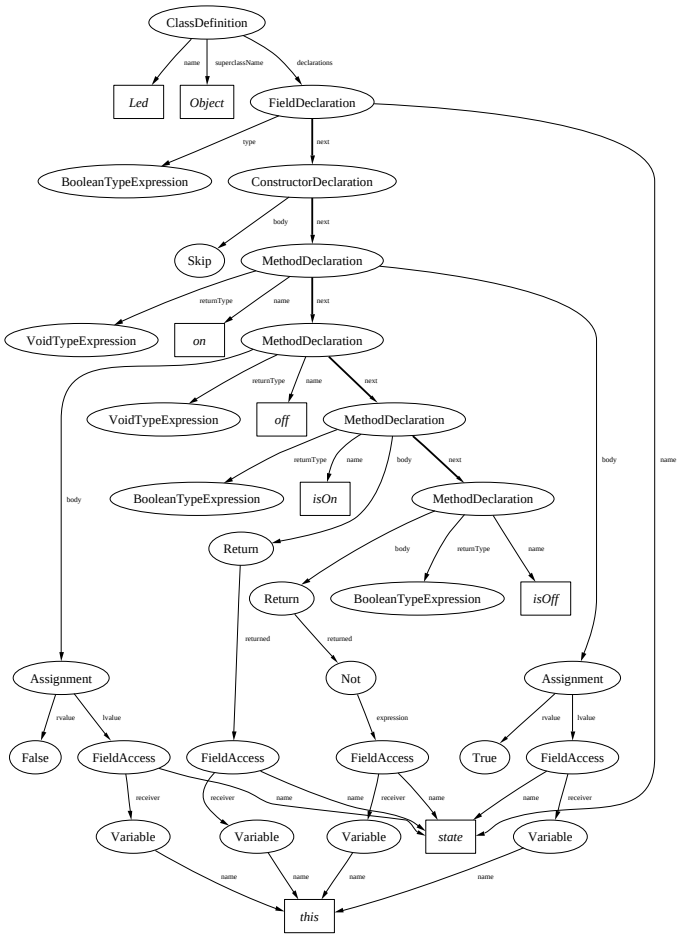


Figura 3.1: La sintassi astratta del programma in Figura 1.4.

Tale struttura logica è chiamata *albero di sintassi astratta* del programma la cui *sintassi concreta* è in Figura 1.4. Vedremo in futuro (Sezione 3.6) come interpretare i nodi di questo albero. Per adesso osserviamo solo che i nodi ovali rappresentano categorie di sintassi astratta composte a partire dai sottoalberi ivi radicati. I nodi rettangolari sono invece gli identificatori del programma e sono condivisi quando occorrono più volte nel codice. In effetti, quindi, sarebbe più corretto parlare di *grafo* di sintassi astratta. Preferiamo comunque continuare a chiamarlo albero dal momento che almeno i nodi ovali sono raggiungibili tramite un unico percorso a partire dalla radice dell'albero (posta in alto). L'albero in Figura 3.1 è più *comodo* da maneggiare da parte di un computer che una sequenza di caratteri come quella in Figura 1.4. Esso infatti è una vera e propria struttura dati i cui diversi livelli di profondità esprimono la struttura del codice sorgente. Questa struttura è invece meno evidente nel codice sorgente.

In questo capitolo parleremo, nell'ordine, delle *grammatiche libere dal contesto* (Sezione 3.1), che sono lo strumento che useremo per descrivere la sintassi del codice sorgente di un programma. Passeremo quindi a descrivere l'uso di uno strumento, chiamato JavaCup, che

permette di creare automaticamente un analizzatore sintattico a partire dalla grammatica del linguaggio che esso deve riconoscere (Sezione 3.2). Quindi analizzeremo il funzionamento di JavaCup, partendo da un modo semplice ma poco potente di effettuare il parsing (Sezione 3.3) e passando quindi a una tecnica più potente (Sezione 3.4). Infine descriveremo la costruzione dell'albero di sintassi astratta da parte di JavaCup (Sezione 3.5) e la struttura dell'albero di sintassi astratta stesso (Sezione 3.6).

3.1 Le grammatiche libere dal contesto

La potenza delle espressioni regolari è limitata poiché esse corrispondono a uno strumento di calcolo con una quantità di memoria limitata a priori (gli *automi a stati finiti* della Sezione 2.6). Esse non sono quindi in grado di esprimere linguaggi la cui definizione richiede la capacità di *contare* fino a livelli di profondità arbitrari, come per esempio il linguaggio

$$P = \{\mathtt{a}^n\mathtt{b}^n \mid n \geq 0\}.$$

Visto il primo carattere `b`, occorre ricordarsi quanti caratteri `a` si sono visti per sapere quanti caratteri `b` ci si deve ancora aspettare. Il *linguaggio di parentesi P* non è un puro gioco teorico: esso astrae un tipico linguaggio di programmazione la cui struttura è data da delimitatori come le parentesi graffe o le parole chiave `begin` ed `end`.

Abbiamo visto nella Sezione 2.9 che le espressioni regolari posso essere *potenziate* utilizzando delle *azioni con memoria* (come l'assegnamento). Ma si tratta di una trucco scomodo per descrivere la complessa sintassi di un linguaggio di programmazione. Meglio è invece identificare uno strumento di descrizione di linguaggi strettamente più potente delle espressioni regolari e naturalmente portato a descrivere la sintassi dei linguaggi di programmazione. Questo strumento sono le *grammatiche libere dal contesto*.

Definizione 11 (Grammatica Libera dal Contesto)**.** Una *grammatica libera dal contesto* (in seguito semplicemente *grammatica*) su un alfabeto Λ è una quadrupla $\langle T, N, I, P\rangle$ dove

- $T \subseteq \Lambda$ è un insieme detto dei *simboli terminali* o semplicemente dei *terminali*
- N è un insieme detto dei *simboli non terminali* o semplicemente dei *non terminali*
- $I \in N$ è il *non terminale iniziale*
- P è un insieme di *produzioni*, cioè di frecce del tipo $L \to \gamma$ dove $L \in N$ è il *lato sinistro* della produzione ed $\gamma \in (T \cup N)^*$ è una sequenza anche vuota di terminali e non terminali, detta *lato destro* della produzione.

Indicheremo in grassetto i terminali, in italico i non terminali e con lettere greche le sequenze, possibilmente vuote, di terminali e non terminali, dette anche *forme sentenziali*. La forma sentenziale vuota sarà sempre indicata come ε. Si noti che il lato destro di una produzione è una forma sentenziale. Una forma sentenziale è detta *ground* o *stringa* se non contiene non terminali.

Un esempio di grammatica è la quadrupla $\langle T, N, I, P\rangle$ dove

$$\begin{aligned} T &= \{\mathsf{a}, \mathsf{b}\} \\ N &= \{I\} \\ P &= \{I \to \varepsilon,\ I \to \mathsf{a}I\mathsf{b}\}. \end{aligned} \tag{3.1}$$

Essa è una grammatica per qualsiasi linguaggio che contenga almeno i simboli a e b. In futuro descriveremo una grammatica semplicemente enumerando le sue produzioni. Assumeremo implicitamente che T è l'insieme dei terminali che occorrono nelle produzioni enumerate e che N è l'insieme dei non terminali che occorrono nelle produzioni enumerate. Assumeremo inoltre che I sia il non terminale a sinistra della prima produzione.

Definizione 12 (Derivazione)**.** Data una grammatica $G = \langle T, N, I, P\rangle$ e due forme sentenziali α e β, diciamo che β è *derivabile in G in un passo* da α se e solo se esiste una produzione $L \to \gamma \in P$ tale che $\alpha = \eta L \delta$ e $\beta = \eta\gamma\delta$. In tal caso scriveremo che $\alpha \Rightarrow_G \beta$. Una *derivazione* per G è la concatenazione di più passi di derivazione $\alpha \Rightarrow \beta_1 \Rightarrow \beta_2 \ldots$ Indicheremo con $\Rightarrow_G^*$ la chiusura riflessiva e transitiva di $\Rightarrow$. Quando G è chiara dal contesto, eviteremo di indicarla esplicitamente nelle notazioni $\Rightarrow$ e $\Rightarrow^*$.

Per esempio, nella grammatica (3.1) si ha

$$\begin{aligned} \mathsf{ab}I\mathsf{b} &\Rightarrow \mathsf{aba}I\mathsf{bb} \\ \mathsf{aba}I\mathsf{bb} &\Rightarrow \mathsf{ababb} \\ I &\Rightarrow \mathsf{a}I\mathsf{b} \\ I &\Rightarrow^* I \\ \mathsf{ab}I\mathsf{b} &\Rightarrow^* \mathsf{ababb}. \end{aligned}$$

Le grammatiche servono a descrivere linguaggi, esattamente come gli automi a stati finiti. In particolare, una grammatica genera il linguaggio formato dalle stringhe derivabili dal non terminale iniziale.

Definizione 13 (Linguaggio generato da una grammatica)**.** Data una grammatica G su un alfabeto Λ, il linguaggio $L(G)$ da essa generato è

$$L(G) = \{\alpha \text{ ground} \mid I \Rightarrow^* \alpha\}.$$

Per esempio, il linguaggio generato dalla grammatica (3.1) è

$$L(G) = \{\mathsf{a}^n\mathsf{b}^n \mid n \geq 0\} = P.$$

Questo mostra che le grammatiche libere dal contesto permettono di generare linguaggi, come P, che non possono essere generati con automi a stati finiti. Più in generale, si può mostrare che ogni linguaggio generabile da un automa a stati finiti è anche generabile da una grammatica libera dal contesto. Conseguentemente, le grammatiche descrivono strettamente più linguaggi che gli automi a stati finiti.

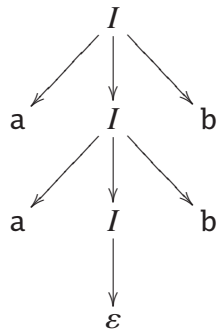


Figura 3.2: Un albero di derivazione per la grammatica (3.1).

Data una forma sentenziale α, potrebbe esserci più di un β tale che $\alpha \Rightarrow \beta$. Per esempio, nella grammatica (3.1) abbiamo $I\mathsf{b} \Rightarrow \mathsf{b}$ ma anche $I\mathsf{b} \Rightarrow \mathsf{a}I\mathsf{bb}$. In questo caso abbiamo la possibilità di scegliere quale produzione usare per sostituire lo stesso non terminale I. In altri casi la pluralità delle scelte è la conseguenza di più occorrenze di non terminali in α. Si consideri per esempio la grammatica

$$\begin{aligned} I &\to \varepsilon \\ I &\to \mathsf{a} \\ I &\to \mathsf{b} \\ I &\to II \end{aligned} \tag{3.2}$$

(come detto sopra, l'enumerazione delle produzioni è sufficiente a definire la grammatica). La stessa stringa abb possiamo derivarla tramite le derivazioni:

$$\begin{aligned} &I \Rightarrow II \Rightarrow I\mathsf{b} \Rightarrow II\mathsf{b} \Rightarrow \mathsf{a}I\mathsf{b} \Rightarrow \mathsf{abb} \\ &I \Rightarrow II \Rightarrow III \Rightarrow II\mathsf{b} \Rightarrow \mathsf{a}I\mathsf{b} \Rightarrow \mathsf{abb} \\ &I \Rightarrow II \Rightarrow I\mathsf{b} \Rightarrow II\mathsf{b} \Rightarrow I\mathsf{bb} \Rightarrow \mathsf{abb}. \end{aligned} \tag{3.3}$$

In questo caso la scelta riguarda l'ordine col quale sostituiamo le I. Va osservato che questo secondo tipo di *libertà di scelta* è in effetti irrilevante, nel senso che, qualsiasi scelta si faccia, è poi possibile effettuare un'altra sostituzione, temporaneamente ritardata. Questa caratteristica delle grammatiche libere dal contesto dice essenzialmente che il criterio con cui si sostituiscono i non terminali in una forma sentenziale non cambia l'insieme delle stringhe derivabili. Potremmo per esempio scegliere indifferentemente di sostituire sempre prima i non terminali più a sinistra (derivazioni *leftmost*) o prima quelli più a destra (derivazioni *rightmost*).

Un modo per astrarre dall'ordine delle sostituzioni è quello di usare degli *alberi di parsing* al posto delle derivazioni stesse. Un albero di parsing rappresenta un *insieme* di derivazioni, tutte quelle che derivano la stessa stringa, a partire dal non terminale di partenza, con qualsiasi criterio di sostituzione.

Definizione 14 (Albero di parsing o di derivazione)**.** Un *albero di parsing* o *di derivazione* per una grammatica $G = \langle T, N, I, P \rangle$ è un albero tale che:

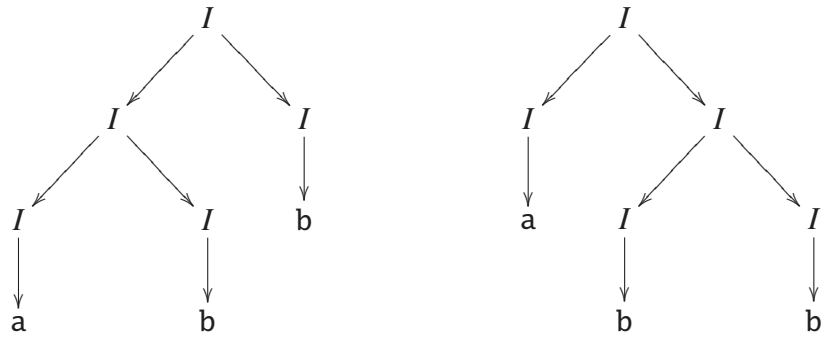


Figura 3.3: Due alberi di derivazione diversi per la stessa stringa abb.

1. i suoi nodi sono etichettati con un elemento di N o di T o con ε;
2. la radice è etichettata con I
3. le foglie sono etichettate con elementi di T o con ε
4. dato un nodo etichettato come L e prese, da sinistra a destra, le etichette $e_1, \ldots, e_n$ dei suoi figli, allora $L \to e_1 \cdots e_n \in P$.

La concatenazione delle etichette della frontiera dell'albero, letta da sinistra a destra secondo una visita leftmost in profondità, è la stringa *derivata* dall'albero a partire dalla sua radice.

Per esempio, la Figura 3.2 mostra un albero di derivazione per la grammatica (3.1). Esso deriva la stringa aabb e astrae l'insieme di derivazioni

$$\{I \Rightarrow \mathtt{a}I\mathtt{b} \Rightarrow \mathtt{aa}I\mathtt{bb} \Rightarrow \mathtt{aabb}\}.$$

L'albero a sinistra nella Figura 3.3 è un albero di derivazione per la grammatica (3.2). Esso deriva la stringa abb e astrae un insieme di derivazioni che include, fra le altre, le derivazioni (3.3).

Gli alberi di derivazione rappresentano insiemi di derivazioni che possiamo considerare equivalenti e interscambiabili. In particolare, un albero di derivazione specifica la struttura logica delle derivazioni che esso rappresenta: come cioè la stringa sulla sua frontiera viene costruita a partire dal non terminale iniziale, senza entrare nei dettagli dell'ordine con cui questa costruzione è effettuata. Ne consegue che, se una stessa stringa α ammette due alberi di derivazione diversi, allora ci sono almeno due modi, *strutturalmente diversi*, di derivare α.

Definizione 15 (Grammatica ambigua)**.** Una grammatica G è *ambigua* se e solo se esiste una forma sentenziale ground α che ammette due alberi di parsing diversi in G.

Per esempio, la grammatica (3.2) è ambigua poiché ammette due alberi di derivazione diversi per la stringa abb, come mostrato in Figura 3.3.

Le grammatiche ambigue ci porranno dei problemi poiché non specificano un'unica struttura per le stringhe di un linguaggio, ma danno la possibilità di interpretarle strutturalmente in modo

diverso. Per esempio, la Figura 3.3 mostra che la stringa `abb` può essere interpretata nella grammatica (3.2) come la stringa `ab` seguita da una `b` (albero a sinistra) o come una `a` seguita dalla stringa `bb` (albero a destra). È quindi importante riuscire a scrivere grammatiche non ambigue, sebbene non esistano vere e proprie regole per farlo. Per esempio, il linguaggio della grammatica ambigua (3.2) è l'insieme di tutte le stringhe di `a` e di `b`. Questa semplice osservazione ci convince che possiamo generare lo stesso linguaggio tramite la grammatica non ambigua:

$$I \to \varepsilon$$
$$I \to \mathtt{a}I$$
$$I \to \mathtt{b}I.$$

Esercizio 11. Si definisca una grammatica sull'alfabeto $\{\mathtt{a}, \mathtt{b}\}$ che genera tutte e sole le sequenze (o liste) non vuote di `a`. Si definisca quindi un'altra grammatica che genera tutte e sole le sequenze, possibilmente vuote, di `a`.

Esercizio 12. Si definisca una grammatica sull'alfabeto $\{\mathtt{a}, \mathtt{b}\}$ che genera tutte e sole le sequenze di `a` e di `b` che contengono tante `a` quante `b`. La grammatica che avete ottenuto è ambigua?

3.2 La generazione dell'analizzatore sintattico di Kitten

Specificheremo la sintassi concreta di Kitten tramite una grammatica libera dal contesto. Si noti che questo non ci permetterà di specificare alcuni aspetti sintattici del linguaggio che non sono specificabili tramite grammatiche libere dal contesto. Per esempio, non saremo capaci di specificare il fatto che un identificare deve essere dichiarato prima di poter essere usato, né l'uso corretto dei tipi nelle espressioni. Questi aspetti verranno quindi considerati *semantici* e gestiti in seguito, in fase, appunto, di analisi semantica (Capitolo 5).

La creazione di un parser a partire dalla grammatica di Kitten verrà ottenuta in maniera automatica, tramite uno strumento Java di nome JavaCup. Esso è una versione Java di un vecchio programma C di nome `yacc` [5]. Dal momento che vogliamo riconoscere il linguaggio Kitten, useremo in JavaCup la grammatica per tale linguaggio. La Figura 3.4 mostra l'utilizzo di JavaCup. L'applicazione di JavaCup alla grammatica del linguaggio Kitten, specificata dentro un file `syntactical/Kitten.cup`, produce tre file:

1. l'analizzatore sintattico vero e proprio `syntactical/Parser.java`. Esso può quindi essere compilato con `javac` ed eseguito per effettuare il parsing di un programma Kitten;

2. il file `syntactical/sym.java`, che enumera i terminali (token) usati dalla grammatica, associando loro un identificatore intero unico; si tratta dello stesso file usato dall'analizzatore lessicale per identificare i token (Sezione 2.3);

3. il file di log `syntactical/Kitten.err`. Tale file contiene eventuali errori nella specifica della grammatica o eventuali problemi riscontrati da JavaCup, quando per esempio non riesce a creare il parser a causa di una grammatica non adeguata. Questo file contiene anche gli stati di un automa di cui parleremo in Sezione 3.4.

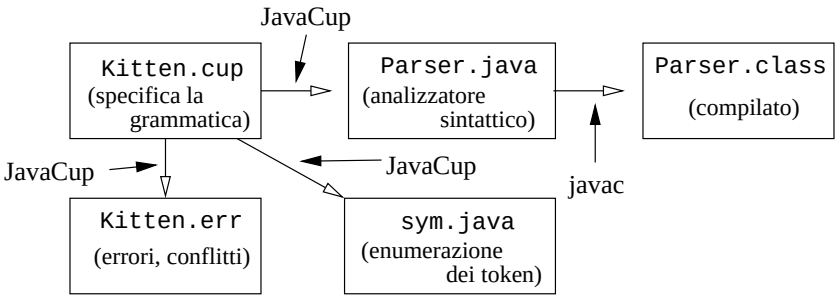


Figura 3.4: La generazione dell'analizzatore sintattico tramite JavaCup.

Se qualcosa non funziona nella generazione del parser dalla grammatica, il programma JavaCup non comunica messaggi di errore da linea di comando ma scrive tutto nel file `syntactical/Kitten.err`. Se quindi, dopo aver modificato o sostituito la grammatica Kitten, JavaCup non genera l'analizzatore `syntactical/Parser.java`, si vada a guardare il contenuto di `syntactical/Kitten.err`.

Vedremo nelle Sezioni 3.3 e 3.4 due modi per generare l'analizzatore sintattico a partire dalla specifica di una grammatica; in particolare, quello descritto nella Sezione 3.4 è quello utilizzato da JavaCup. Nella Sezione 3.5 vedremo come usare JavaCup per generare anche l'albero che descrive la struttura del file sorgente, come quello mostrato in Figura 1.4. Per adesso descriviamo la specifica della grammatica Kitten contenuta nel file `syntactical/Kitten.cup`.

3.2.1 La specifica dei terminali e dei non terminali

Il file `syntactical/Kitten.cup` inizia con l'enumerazione dei terminali della grammatica:

```
terminal ID, STRING, INTEGER, FLOATING,
         CLASS, EXTENDS, FIELD, METHOD, CONSTRUCTOR, NEW,
         INT, FLOAT, BOOLEAN, VOID,
         COMMA, SEMICOLON, AS, LPAREN, RPAREN,
         LBRACK, RBRACK, LBRACE, RBRACE, DOT, PLUS, MINUS,
         TIMES, DIVIDE, EQ, NEQ, LT, LE, GT, GE, AND, OR, NOT,
         ASSIGN, ARRAY, IF, THEN, ELSE, WHILE, FOR,
         OF, RETURN, NIL, TRUE, FALSE, UMINUS;
```

Si noti che essi, tranne `UMINUS` (Sezione 3.2.4), sono esattamente gli stesso terminali (token) generati dall'analizzatore lessicale del Capitolo 2. L'ordine con cui essi sono enumerati non ha normalmente importanza, ma ritorneremo su questo aspetto. Segue quindi una enumerazione dei non terminali[1] della grammatica:

[1]Non scriviamo i non terminali in italico poiché si tratta di un file testo che viene passato a JavaCup.

```
non terminal class;
non terminal class_members;
non terminal formals;
non terminal formals_aux;
non terminal statements;
non terminal command;
non terminal exp;
non terminal expseq;
non terminal expseq_aux;
non terminal lvalue;
non terminal type;
non terminal typeplus;
```

Un ben preciso non terminale è marcato come non terminale iniziale della grammatica:

```
start with class;
```

Questo significa che un programma Kitten deve essere derivabile a partire dal non terminale `class` ovvero, per come daremo le produzioni di `class`, che un programma Kitten non è altro che una definizione di una classe, che eventualmente usa altre classi definite in altri file sorgenti.

Seguono a questo punto le produzioni della grammatica per ogni non terminale del linguaggio enumerato sopra. Li commenteremo secondo un ordine bottom-up, partendo cioè da tipi, espressioni e comandi e andando verso il non terminale per la classe.

3.2.2 La specifica dei tipi Kitten

I tipi Kitten sono descritti dalle seguenti produzioni:

```
type ::=
   ID
 | BOOLEAN
 | INT
 | FLOAT
 | ARRAY OF type ;
```

La barra verticale indica un'alternativa nella definizione di `type`. Si noti che oltre ai tre tipi primitivi `boolean`, `int` e `float`, sono previsti dei tipi classe, espressi come il token `ID`, nonché il tipo array, definito ricorsivamente. Il non terminale `type` è usato quando si vuole specificare il tipo di una variabile, per esempio in una dichiarazione, o in una espressione, come in un cast. Se vogliamo invece specificare il tipo di ritorno di un metodo useremo il non terminale `typeplus` che ammette anche il tipo `void`:

```
typeplus ::=
   type
 | VOID ;
```

3.2.3 La specifica delle espressioni Kitten

Un'espressione (Sezione 1.7) è definita in maniera ricorsiva tramite le seguenti produzioni:

```
exp ::=
   lvalue                            // leftvalue
 | TRUE                              // la costante true
 | FALSE                             // la costante false
 | INTEGER                           // una costante intera
 | FLOATING                          // una costante in virgola mobile
 | STRING                            // una costante stringa fra apici
 | NIL                               // il riferimento costante nil
 | NEW ID LPAREN expseq RPAREN       // la creazione di un oggetto
 | NEW type LBRACK exp RBRACK        // la creazione di un array
 | exp AS type                       // un cast o conversione di tipo
 | exp PLUS exp                      // un'addizione
 | exp MINUS exp                     // una sottrazione
 | exp TIMES exp                     // una moltiplicazione
 | exp DIVIDE exp                    // una divisione
 | MINUS exp                         // l'opposto di un valore intero
 | exp LT exp                        // <
 | exp LE exp                        // <=
 | exp GT exp                        // >
 | exp GE exp                        // >=
 | exp EQ exp                        // uguaglianza
 | exp NEQ exp                       // disuguaglianza
 | exp AND exp                       // and logico
 | exp OR exp                        // or logico
 | NOT exp                           // negazione logica
 | exp DOT ID LPAREN expseq RPAREN // chiamata di metodo non void
 | LPAREN exp RPAREN ;               // parentesi
```

Tali produzioni dicono in primo luogo che i *leftvalue* sono espressioni. In effetti, tutte le espressioni possono essere usate alla destra di un assegnamento (come *rightvalue*) ma una categoria ristretta di espressioni, dette *leftvalue*, può essere usata *anche* alla sinistra di un assegnamento. A tal fine, tali espressioni devono fare riferimento a una ben precisa cella di memoria dentro la quale l'assegnamento può scrivere. Ci sono solo tre tipi di leftvalue in Kitten:

1. le variabili, come in `a := 35`; in questo caso l'assegnamento modifica la cella di memoria che contiene il valore della variabile `a`;
2. i campi degli oggetti, come in `o.f := a`; in questo caso l'assegnamento modifica la cella di memoria che contiene il valore del campo `f` dell'oggetto contenuto nella variabile `o`;
3. gli elementi degli array, come in `arr[i] := a`; in questo caso l'assegnamento modifica la cella *i*-esima dell'array `arr`.

Conseguentemente, la specifica dei leftvalue è data tramite le seguenti produzioni:

```
lvalue ::=
   ID                                  // una variabile
 | exp DOT ID                          // il campo di un oggetto
 | exp LBRACK exp RBRACK ;             // un elemento di un array
```

Si noti che la sintassi di espressioni e leftvalue è mutuamente ricorsiva.

Le precedenti produzioni per `exp` dicono anche che tutte le costanti del linguaggio sono espressioni: cioè `true`, `false`, le costanti intere, le costanti in virgola mobile, le stringhe fra doppi apici e il riferimento `nil`. È un'espressione anche la creazione di un oggetto come in `new Rettangolo(10,20)`. Questo è ottenuto tramite la produzione per `exp`:

```
exp ::= NEW ID LPAREN expseq RPAREN
```

dove `ID` è l'identificatore della classe che si sta istanziando e `expseq` è una sequenza, possibilmente vuota, di espressioni separate da virgole:

```
expseq ::=
 | expseq_aux ;

expseq_aux ::=
   exp
 | exp COMMA expseq_aux ;
```

Si noti che la prima produzione per `expseq` è una ε-produzione.

La creazione di un array specifica sia il tipo dei suoi elementi che la dimensione dell'array, come in `new int[50]`, ed è ottenuta tramite la produzione

```
exp ::= NEW type LBRACK exp RBRACK
```

Kitten non ammette la creazione diretta di array multidimensionali. Essi devono quindi essere creati a partire da un array monodimensionale i cui elementi sono a loro volta degli array che vanno essi stessi creati esplicitamente come array monodimensionali.

Il cast viene effettuato in Kitten con la notazione `as`, come in `persona as Studente`. Questa sintassi è resa possibile dalla produzione:

```
exp ::= exp AS type
```

Si noti che `type` non è vincolato in alcun modo (tranne a non essere `void`, infatti abbiamo usato `type` e non `typeplus`). È quindi del tutto lecito scrivere `12 as float` oppure `arr as array of int`. In particolare, la stessa notazione viene usata sia per le conversioni di tipo (`12 as float`) che per i cast veri e propri (`persona as Studente` e `arr as array of int`).

Le precedenti produzioni per le espressioni includono le operazioni binarie, sia aritmetiche (addizione, sottrazione, moltiplicazione e divisione) che di confronto (maggiore, maggiore o uguale, minore, minore o uguale, uguale, diverso) che logiche (and logico e or logico). Infine includono le due operazioni unarie di negazione di interi (`MINUS`) e di valori logici (`NOT`) e l'espressione per la chiamata di metodo che non ritorna `void` (Sezione 1.7), ottenuta con la produzione:

```
exp ::= exp DOT ID LPAREN expseq RPAREN
```

in cui, come si può vedere, è richiesta la presenza esplicita del ricevitore della chiamata.

Infine, una espressione fra parentesi tonde viene ancora considerata come un'espressione. Questo permette per esempio di cambiare la precedenza degli operatori aritmetici rispetto a quella di default, descritta nella prossima sezione.

3.2.4 La specifica della precedenza degli operatori aritmetici

La grammatica per le espressioni aritmetiche della sezione precedente è chiaramente ambigua. Per esempio, l'espressione `2 + a * 4` può essere interpretata sia come `(2 + a) * 4` che come `2 + (a * 4)`. Abbiamo già anticipato che le grammatiche ambigue ci porranno problemi in fase di creazione dell'analizzatore sintattico, il quale essenzialmente non sa quale delle due interpretazioni preferire. Sappiamo dall'esperienza con altri linguaggi di programmazione che la moltiplicazione ha *precedenza* rispetto all'addizione, per cui l'interpretazione da preferire è `2 + (a * 4)`. Un problema simile sorge di fronte ad espressioni come `2 - a - 4` che sono normalmente intese come `(2 - a) - 4` piuttosto che come `2 - (a - 4)`. Si noti che queste due interpretazioni sono diverse poiché, se `a` valesse 0, la prima interpretazione darebbe a `2 - a - 4` il valore -2 mentre la seconda le darebbe il valore 6. In questo caso si tratta di un problema di *associatività* degli operatori e il fatto che l'interpretazione da preferire sia `(2 - a) - 4` significa che gli operatori aritmetici normalmente *associano a sinistra*. Queste regole vanno in qualche modo specificate nella grammatica.

Un primo approccio è quello di individuare una grammatica non ambigua che dà priorità alla moltiplicazione (e divisione) rispetto all'addizione (e sottrazione) e che specifica l'associatività a sinistra degli operatori. Per esempio si potrebbero usare le seguenti produzioni al posto di quelle della Sezione 3.2.3:

```
exp ::=
   term
 | exp PLUS term
 | exp MINUS term ;
```

la quale definisce una `exp` come una lista non vuota di `term`, separati da + o -, con associatività a sinistra. Il nuovo non terminale `term` viene definito a sua volta come una lista non vuota di `factor`, separati da * e /, con associatività a sinistra:

```
term ::=
   factor
 | term TIMES factor
 | term DIVIDE factor ;
```

dove

```
factor ::= ... altre produzioni per le espressioni ...
```

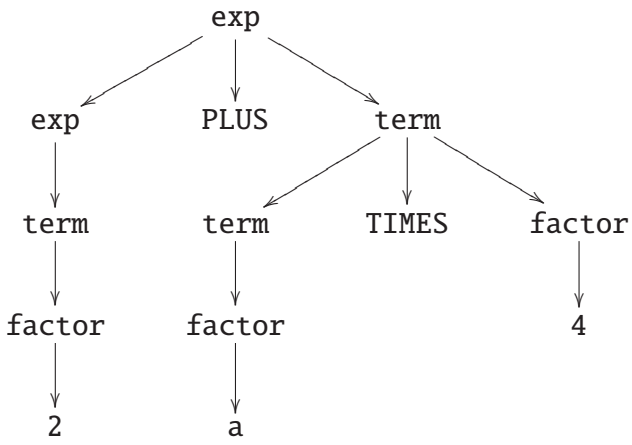


Figura 3.5: Un albero di derivazione per `2 + a * 4` tramite una grammatica non ambigua per le espressioni aritmetiche.

Con queste produzioni, l'unico albero di derivazione possibile per `2 + a * 4` a partire da `exp` è quello mostrato in Figura 3.5, dove le foglie etichettate come `2`, `a` e `4` vanno intese come `INTEGER`, `ID` ed `INTEGER`, rispettivamente, ma abbiamo preferito indicare il loro valore lessicale per maggiore chiarezza.

L'uso di una grammatica non ambigua, sebbene possibile, risulta però scomodo perché, come si vede, occorre strutturare la grammatica in una maniera non molto intuitiva. Non sarà quindi questa la strada seguita in Kitten. Risulta più semplice infatti lasciare ambigua la grammatica ma specificare come risolvere l'ambiguità tramite delle regole di priorità e associatività per gli operatori binari. In particolare, basta aggiungere le seguenti dichiarazioni all'interno del file `syntactical/Kitten.cup`:

```
precedence left AND, OR;
precedence left NOT;
precedence nonassoc EQ, NEQ, LT, LE, GT, GE;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence left UMINUS;
precedence left DOT, LBRACK;
```

Queste dichiarazioni enumerano gli operatori di Kitten in ordine di priorità crescente. Esse indicano inoltre la loro associatività, la qual cosa ha senso comunque solo per gli operatori binari. L'effetto di queste dichiarazioni è in primo luogo quello di dare priorità minima agli operatori logici binari e priorità subito maggiore alla negazione logica. Per esempio, questo fa sì che `!a & b` venga interpretato come `(!a) & b`. L'associatività è richiesta sinistra, così che per esempio `a & b | c` viene interpretato come `(a & b) | c`. Gli operatori di confronto hanno priorità superiore a quelli logici, in modo tale che `a & b = c` venga interpretato come `a & (b = c)`. Inoltre è richiesto che essi non siano associativi. Questo significa che viene considerato come

un errore scrivere `a = b = c`. La priorità successiva è quella degli operatori aritmetici, con addizioni e sottrazioni che hanno priorità inferiore a moltiplicazioni e divisioni. Ancora maggiore è la priorità del token `UMINUS`. Si noti che tale token non viene mai ritornato dall'analizzatore lessicale del Capitolo 2. Esso è un token fittizio usato solo per avere una priorità maggiore della moltiplicazione e della divisione. Tale priorità viene data esplicitamente alla produzione

```
exp ::= MINUS exp %prec UMINUS
```

per le espressioni. Questo fa sì per esempio che l'espressione `-a * b` venga interpretato come `(-a) * b`. L'ultima precedenza dà al punto e alla parentesi quadra aperta una priorità maggiore di quella di tutti gli altri operatori. Per esempio, questo permette di interpretare l'espressione `a < b.f` come `a < (b.f)` piuttosto che come `(a < b).f` ed `a < b[5]` come `a < (b[5])` piuttosto che come `(a < b)[5]`.

Nella Sezione 3.4.5 vedremo come queste regole di priorità e associatività vengono utilizzate da JavaCup per risolvere l'ambiguità della grammatica.

3.2.5 La specifica dei comandi Kitten

La specifica dei comandi è ricorsiva e utilizza quella delle espressioni (Sezione 1.5). La parte di grammatica che specifica l'insieme dei comandi è formata dalle produzioni:

```
command ::=
   lvalue ASSIGN exp  // un assegnamento
 | type ID ASSIGN exp // una dichiarazione di variabile
 | RETURN             // un return da un metodo che ritorna void
 | RETURN exp         // un return da un metodo che non ritorna void
 | IF LPAREN exp RPAREN THEN command  // un if/then
 | IF LPAREN exp RPAREN THEN command
     ELSE command                    // un if/then/else
 | WHILE LPAREN exp RPAREN command    // un ciclo while
 | FOR LPAREN command SEMICOLON exp
     SEMICOLON command RPAREN command // un ciclo for
 | LBRACE statements RBRACE           // uno scope locale
 | LBRACE RBRACE                      // un comando vuoto
 | exp DOT ID LPAREN expseq RPAREN ;  // una chiamata di metodo
```

Osserviamo che queste produzioni lasciano facoltativi l'`else` di un `if` e l'espressione di ritorno di un comando `return`. Inoltre esse non impongono in alcun modo che dentro un metodo che ritorna `void` si trovino solo `return` *senza* espressione di ritorno, né che dentro un metodo che non ritorna `void` si trovino `return` *con* un'espressione di ritorno. Questi vincoli complicherebbero notevolmente la grammatica. Inoltre non riusciremmo comunque a controllare che i tipi ritornati corrispondano a quelli dichiarati per i metodi. Preferiamo invece lasciare questi (e altri) controlli alla successiva fase di analisi semantica (Capitolo 5).

La produzione

```
exp ::= LBRACE statements RBRACE
```

definisce uno *scope locale*, cioè una sequenza non vuota di comandi, separati da punti e virgola, le cui dichiarazioni restano locali a tale sequenza stessa (ovvero non sono più visibili dopo la parentesi graffa di chiusura). Il non terminale `statements` definisce proprio tale sequenza di comandi:

```
statements ::=
   command
 | command SEMICOLON statements ;
```

3.2.6 La specifica di una classe Kitten

Definiti comandi ed espressioni, possiamo finalmente definire la sintassi di una classe, cioè del non terminale di partenza della grammatica Kitten (Sezione 3.2.1). La definizione è la seguente:

```
class ::=
   CLASS ID LBRACE class_members RBRACE
 | CLASS ID EXTENDS ID LBRACE class_members RBRACE ;
```

Come si può vedere, l'indicazione della superclasse è facoltativa. Quando manca, si assume implicitamente che essa sia `Object`. I membri della classe sono una lista possibilmente vuota di dichiarazioni di campi, costruttori e metodi:

```
class_members ::=
 | FIELD type ID class_members                // un campo
 | CONSTRUCTOR LPAREN formals RPAREN command
     class_members                            // un costruttore
 | METHOD typeplus ID LPAREN formals RPAREN command
     class_members ;                          // un metodo
```

La prima produzione per `class_members` è una ε-produzione, in modo che una lista di membri di una classe possa anche essere vuota. Osserviamo che il corpo dei metodi e dei costruttori è un comando, senza obbligo di presenza delle parentesi graffe intorno al corpo. Inoltre il tipo di ritorno di un metodo è un `typeplus`, in modo da ammettere anche `void` (Sezione 3.2.2). I parametri formali di costruttori e metodi non sono altro che una lista possibilmente vuota di coppie tipo/nome del parametro, separate da virgola:

```
formals ::=
 | formals_aux ;

formals_aux ::=
   type ID
 | type ID COMMA formals_aux ;
```

Si noti che la prima produzione per `formals` è una ε-produzione.

3.2.7 L'interfaccia con l'analizzatore lessicale

Specificata la grammatica, occorre ancora indicare, in `syntatical/Kitten.cup`, come interfacciarsi con l'analizzatore lessicale che abbiamo ottenuto nel Capitolo 2. A tal fine, inseriamo fra i delimitatori `parser code {:` e `:};` del codice che verrà riportato testualmente all'inizio dell'analizzatore sintattico generato da JavaCup. Tale codice si preoccupa di definire un campo dell'analizzatore sintattico di tipo `lexical.Lexer` e di inizializzarlo all'interno del suo costruttore con un nuovo analizzatore lessicale. Inoltre, esso fa sì che si possano segnalare degli errori di sintassi tramite la struttura di errore associata all'analizzatore lessicale appena creato:

```
parser code {:
  private Lexer lexer;

  public Parser(Lexer lexer) {
    this.lexer = lexer;
  }

  public ErrorMsg getErrorMsg() {
    return lexer.getErrorMsg();
  }

  public void syntax_error(java_cup.runtime.Symbol token) {
    getErrorMsg().error(token.left,"syntax error");
  }
:};
```

Il metodo `syntax_error()` viene chiamato dall'analizzatore sintattico quando non riesce ad effettuare il parsing di un programma. Il metodo `getErrorMsg()` lo usiamo invece per ottenere la struttura di errore con cui comunicare messaggi di errore relativi al file sorgente che stiamo processando con un dato analizzatore sintattico. Tale struttura dati è la stessa che abbiamo usato per fare l'analisi lessicale dello stesso file sorgente. Tale metodo ci sarà utile in futuro quando vorremo segnalare degli errori in fasi successive di compilazione.

L'ultimo passo è di definire in che modo l'analizzatore sintattico ottiene i token del programma sorgente. Specifichiamo di utilizzare il metodo `nextToken()` dell'analizzatore lessicale (Sezione 2.3):

```
scan with {:
  return lexer.nextToken();
:};
```

Si noti che avremmo potuto utilizzare un qualsiasi altro nome al posto di `nextToken()`, purché sia lo stesso utilizzato dall'analizzatore lessicale.

Il risultato della trasformazione di `syntactical/Kitten.cup` in un parser è una classe Java `syntactical/Parser.java` che contiene un metodo `parse()`. Chiamando tale metodo si effettua il parsing del file sorgente, segnalando eventuali errori di sintassi.

$$
\begin{aligned}
I &\rightarrow com\ \$ \\
com &\rightarrow exp\ \texttt{ASSIGN}\ \texttt{INTEGER} \\
exp &\rightarrow \texttt{ID} \\
exp &\rightarrow \texttt{INTEGER} \\
exp &\rightarrow \texttt{MINUS}\ exp
\end{aligned}
$$

Figura 3.6: Una grammatica *LL*(1).

Ricordiamo infine che, come mostrato in Figura 3.4, il programma JavaCup genera anche il file `syntactical/sym.java` che contiene una enumerazione dei terminali della grammatica. Tale enumerazione viene usata anche dall'analizzatore lessicale (Sezione 2.3).

Il fatto che il file `syntactical/sym.java` venga generato da JavaCup insieme all'analizzatore sintattico che ne fa uso e che a sua volta contiene quello lessicale, il quale usa anch'esso `syntactical/sym.java`, pone un fastidioso problema di ciclicità nei casi in cui si vuole modificare l'insieme dei token del linguaggio Kitten. Occorre allora aggiungere il token fra le espressioni regolari di `lexical/Lexer.java` nonché, manualmente, nell'enumerazione in `syntactical/sym.java`, dandogli un codice arbitrario ma non usato dagli altri token. A questo punto l'analizzatore lessicale e quello sintattico possono essere compilati. La compilazione di quest'ultimo genera però un *nuovo* file `syntactical/sym.java`, che enumera i token in modo possibilmente diverso dall'enumerazione manuale che avevamo appena dato. Occorre ricompilare l'analizzatore lessicale per far sì che i due analizzatori siano finalmente sincronizzati sulla stessa enumerazione. In termini di `make`, queste due compilazioni si possono ottenere con `make clean; make syntactical; make clean; make syntactical`.

3.3 Il parsing *LL*

Descriviamo in questa sezione il modo più semplice, ma purtroppo anche meno potente, per costruire un analizzatore sintattico a partire da una grammatica. Si consideri la grammatica in Figura 3.6. La prima produzione indica che un file sorgente che rispetta le regole di questa grammatica deve contenere un *com* seguito da un carattere `$` che indica la fine del file sorgente. Quando una grammatica contiene solo una produzione per il non terminale iniziale I, la quale inoltre termina con il token `$`, si dice che essa è *aumentata*[2]. Il senso è che questa produzione impone che non ci siano caratteri in più dopo il *com*.

Supponiamo quindi di essere posizionati all'inizio del file sorgente. Affinché il sorgente sia corretto, davanti a noi deve esserci un *com* e poi la fine del file. Supponiamo di essere posizionati

[2]L'aumento di una grammatica è ottenuto in JavaCup tramite la direttiva `start with`. In questo esempio con `start with com`.

```
import syntactical.sym;

public class Parser {
  private Lexer lexer;
  private java_cup.runtime.Symbol lookahead;

  public Parser(Lexer lexer) {
    this.lexer = lexer; lookahead = lexer.nextToken();
  }
  ... syntax_error ...
  private void eat(java_cup.runtime.Symbol token) {
    if (lookahead.sym != token.sym) syntax_error(lookahead);
    lookahead = lexer.nextToken();
  }
  public void parse() { parseI(); }
  private void parseI() { parseCom(); eat(sym.EOF); }
  private void parseCom() { parseExp(); eat(sym.ASSIGN); eat(sym.INTEGER); }
  private void parseExp() {
    switch (lookahead.sym) {
      case sym.ID: eat(sym.ID); break;
      case sym.INTEGER: eat(sym.INTEGER); break;
      case sym.MINUS: eat(sym.MINUS); parseExp();
      default: syntax_error(lookahead);
    }
  }
}
```

Figura 3.7: L'implementazione in Java di un parser *LL*(1) per la grammatica in Figura 3.6.

da qualche parte all'interno del file sorgente e di dovere riconoscere un *com*. Per come è scritta la grammatica, l'unica speranza è che dinanzi a noi ci sia un *exp* seguito dal token `ASSIGN` seguito dal token `INTEGER`. Supponiamo infine di essere posizionati da qualche parte nel file sorgente e di dovere riconoscere un *exp*. Questa volta abbiamo tre possibilità, mutuamente esclusive:

1. davanti a noi c'è il token `ID`
2. oppure davanti a noi c'è il token `INTEGER`
3. o infine davanti a noi c'è il token `MINUS` e siamo poi capaci di riconoscere, ricorsivamente, un altro *exp*.

Questo semplice ragionamento ci permette di scrivere dei metodi privati dell'analizzatore sintattico, *uno per ogni non terminale*, i quali si occupano di riconoscere il corrispondente non terminale della grammatica. La Figura 3.7 mostra l'implementazione in Java di tali metodi. Si noti che si tratta di un raffinamento del codice che abbiamo visto nella Sezione 3.2.7. Il metodo

ausiliario `eat()` impone che davanti a noi ci sia il token indicato come parametro. In caso contrario segnala un errore di sintassi. L'implementazione di `parseI()`, `parseCom()` e `parseExp()` ricalca il ragionamento fatto sopra. Si noti in particolare che `parseExp()` è ricorsivo e che il suo comportamento è guidato dal token, detto *lookahead*, che ci troviamo davanti al momento della sua chiamata. Se tale token non è nessuno dei tre leciti, segnaliamo un errore di sintassi. Affinché il file sorgente sia corretto, occorre poter riconoscere il non terminale I all'inizio di tale file. Questo è proprio quello che facciamo nel metodo `parse()` che effettua il parsing del file sorgente.

La generazione del codice in Figura 3.7 può essere fatta in maniera automatica a partire dalla grammatica in Figura 3.6 poiché i metodi del tipo `parse`N ricalcano direttamente l'insieme delle produzioni per il non terminale N. L'unica richiesta è che sia possibile, per ogni non terminale definito da più di una produzione, decidere quale produzione utilizzare sulla base del token che si ha di fronte. Per esempio, il non terminale *exp* è definito da tre produzioni in Figura 3.6 ed è sempre possibile decidere quale delle tre utilizzare guardando il token che ci sta davanti, come mostrato nell'implementazione in Figura 3.7. Il motivo è che le tre produzioni hanno lati destri che *iniziano* con token distinti, per cui non c'è alcun token che dà origine ad ambiguità su quale delle tre produzioni applicare. Questo ragionamento ci dice che per costruire un programma come quello in Figura 3.7 occorre conoscere, per ogni non terminale definito dalle produzioni $L \to r_1, \ldots, L \to r_n$ con $n \geq 2$, qual è l'insieme dei token con cui può cominciare ciascuno dei lati destri $r_1, \ldots, r_n$ e occorre che tali insiemi siano disgiunti. Per esempio, il lato destro della produzione `MINUS` *exp* in Figura 3.6 inizia con l'insieme di token {`MINUS`}.

Calcolare questi inizi non è così semplice come può sembrare a prima vista, dal momento che i lati destri delle produzioni potrebbero cominciare con un non terminale, come in *exp* $\to$ *exp* `PLUS` *exp*, o potrebbero essere *annullabili*, come in *exp* $\to \varepsilon$, nel qual caso non c'è alcun token con cui essi iniziano ed occorre invece guardare quali token potrebbero *seguire* il lato sinistro della produzione, in questo caso *exp*, per capire quando selezionare la produzione. Affrontiamo questi problemi formalmente nella sezione seguente.

3.3.1 Gli insiemi nullable, first e follow.

Abbiamo accennato al fatto che una ε-produzione del tipo *exp* $\to \varepsilon$ deve essere trattata con cura nella costruzione del parsing. Più in generale, la stessa attenzione deve essere usata per tutte quelle produzioni il cui lato destro è *annullabile*, cioè è una forma sentenziale da cui è possibile derivare ε.

Definizione 16. Data una grammatica G e una forma sentenziale α, diciamo che α è *annullabile in* G se e solo se $\alpha \Rightarrow_G^* \varepsilon$. L'insieme dei non terminali annullabili di G è indicato come $\mathsf{nullable}_G$ (di solito omettiamo G se essa è chiara dal contesto).

Si noti che basta conoscere i non terminali annullabili per poter calcolare, in maniera composizionale, l'annullabilità di una qualsiasi altra forma sentenziale.

Proposizione 1. Sia $\langle T, N, I, P\rangle$ una grammatica e $\nu \subseteq N$. Sia α una forma sentenziale e definiamo $\mathsf{nullable}(\alpha, \nu)$ come segue:

$$\begin{aligned} \mathsf{nullable}(\varepsilon, \nu) &= \mathit{true} \\ \mathsf{nullable}(t, \nu) &= \mathit{false} \quad \text{per ogni } t \in T \\ \mathsf{nullable}(n, \nu) &= (n \in \nu) \quad \text{per ogni } n \in N \\ \mathsf{nullable}(\alpha_1\alpha_2, \nu) &= \mathsf{nullable}(\alpha_1, \nu) \wedge \mathsf{nullable}(\alpha_2, \nu). \end{aligned} \tag{3.4}$$

Se ν è l'insieme dei non terminali annullabili in G, allora α è annullabile se e solo se $\mathsf{nullable}(\alpha, \nu)$. In tal caso scriviamo $\mathsf{nullable}(\alpha)$ piuttosto che $\mathsf{nullable}(\alpha, \nu)$.

Il calcolo dell'insieme nullable per una grammatica si effettua come un calcolo di punto fisso. Partendo da un insieme vuoto ϕ^0 di annullabili, si aggiungono prima i nonterminali che hanno una produzione con ε come lato destro, ottenendo ϕ^1. Quindi, se $\phi^1 \neq \phi^0$, si aggiungono i non terminali che hanno alla destra solo non terminali e tutti annullabili poiché in ϕ^1. Si ottiene così ϕ^2. Il procedimento viene iterato finché non si raggiunge un punto fisso, cioè un ϕ^k tale che $\phi^k = \phi^{k+1}$.

Proposizione 2. Sia $\langle T, N, I, P\rangle$ una grammatica e $\nu \subseteq N$. Definiamo

$$\phi(\nu) = \{L \mid L \to r \in P \text{ e } \mathsf{nullable}(r, \nu)\}.$$

Si ha $\mathsf{nullable}_G = \mathit{lfp}(\phi)$, dove $\mathit{lfp}(\phi)$ è il minimo punto fisso di ϕ, calcolabile come $\lim_{i\to\infty} \phi^i$, con

$$\begin{aligned} \phi^0 &= \emptyset \\ \phi^{i+1} &= \phi(\phi^i) \quad \text{per ogni } i \geq 0. \end{aligned}$$

Questo risultato può sembrare complesso ma è in realtà di facile applicazione. Essenzialmente dice di cominciare il calcolo di nullable dall'insieme vuoto e di aggiungervi man mano quei non terminali che stanno alla sinistra di una produzione il cui lato destro risulta annullabile secondo le equazioni (3.4). Questo calcolo deve essere iterato finché non è più possibile aggiungere nuovi non terminali. Si noti che l'uso di ∞ nel limite della Proposizione 2 non significa che serve un numero infinito di iterazioni, poiché l'insieme dei non terminali di una data grammatica è finito e quindi tale limite verrà sicuramente raggiunto in un numero di iterazioni pari, *al massimo*, al numero dei non terminali della grammatica.

Vediamo un primo esempio. Vogliamo calcolare i non terminali annullabili della grammatica in Figura 3.6. Cominciamo dall'insieme vuoto, che scriviamo con una tabella che dice che nessun non terminale è al momento considerato annullabile:

nullable

	ϕ^0
I	*false*
com	*false*
exp	*false*

$$
\begin{aligned}
I &\to L\$ \\
L &\to AB \\
A &\to \varepsilon \\
A &\to \mathtt{a}A \\
B &\to \varepsilon \\
B &\to \mathtt{b}B
\end{aligned}
$$

Figura 3.8: Un'altra grammatica *LL*(1).

A questo punto dobbiamo considerare tutte le produzioni della grammatica e marcare come annullabili i non terminali alla sinistra di una produzione il cui lato destro risulta annullabile secondo le equazioni (3.4). Dal momento che tutti i lati destri in Figura 3.6 contengono almeno un terminale, nessuno di essi è annullabile e otteniamo

nullable

	ϕ^0	ϕ^1
I	*false*	*false*
com	*false*	*false*
exp	*false*	*false*

(3.5)

Basta osservare che $\phi^0 = \phi^1$ per concludere che abbiamo già raggiunto il punto fisso, per cui $\mathsf{nullable} = \phi^1 = \emptyset$.

Si consideri adesso la grammatica in Figura 3.8. Ancora una volta, partiamo dall'insieme vuoto di non terminali annullabili:

nullable

	ϕ^0
I	*false*
L	*false*
A	*false*
B	*false*

A questo punto consideriamo le produzioni in Figura 3.8. Il lato destro di $I \to L\$$ non è annullabile in ϕ^0 poiché contiene il terminale \$:

$$\mathsf{nullable}(L\$, \phi^0) = \mathit{false}.$$

Il lato destro di $L \to AB$ non è annullabile in ϕ^0 poiché nessuno dei due non terminali A e B è annullabile in ϕ^0:

$$\mathsf{nullable}(AB, \phi^0) = \mathit{false}.$$

Il lato destro di $A \to \varepsilon$ è annullabile in ϕ^0:

$$\mathsf{nullable}(\varepsilon, \phi^0) = \mathit{true}$$

così come il lato destro di $B \to \varepsilon$. I lati destri di $A \to \mathsf{a}A$ e $B \to \mathsf{b}B$ non sono annullabili in ϕ^0 dal momento che contengono un terminale:

$$\mathsf{nullable}(\mathsf{a}A, \phi^0) = \mathsf{nullable}(\mathsf{b}B, \phi^0) = \mathit{false}.$$

Concludiamo che ϕ^1 è come indicato in questa tabella:

nullable

	ϕ^0	ϕ^1
I	*false*	*false*
L	*false*	*false*
A	*false*	*true*
B	*false*	*true*

Dal momento che $\phi^0 \neq \phi^1$, non abbiamo ancora raggiunto il punto fisso e dobbiamo calcolare ϕ^2. L'unica differenza rispetto al calcolo di ϕ^1 è che adesso

$$\mathsf{nullable}(AB, \phi^1) = \mathit{true}$$

dal momento che sia A che B sono annullabili in ϕ^1. Concludiamo che ϕ^2 è come indicato nella seguente tabella:

nullable

	ϕ^0	ϕ^1	ϕ^2
I	*false*	*false*	*false*
L	*false*	*false*	*true*
A	*false*	*true*	*true*
B	*false*	*true*	*true*

Dal momento che $\phi^1 \neq \phi^2$ dobbiamo calcolare ϕ^3. Il suo calcolo risulta identico a quello di ϕ^2 per cui in definitiva otteniamo:

nullable

	ϕ^0	ϕ^1	ϕ^2	ϕ^3
I	*false*	*false*	*false*	*false*
L	*false*	*false*	*true*	*true*
A	*false*	*true*	*true*	*true*
B	*false*	*true*	*true*	*true*

(3.6)

ovvero $\phi^2 = \phi^3$ è il punto fisso cercato e quindi $\mathsf{nullable} = \{L, A, B\}$.

La funzione ϕ della Proposizione 2 è monotona crescente. Conseguentemente, l'insieme dei non terminali considerati annullabili non può che crescere durante il calcolo del punto fisso. Questa osservazione, che è alla base dell'esistenza stessa del punto fisso cercato, implica anche che se durante il calcolo di una tabella per nullable vediamo decrescere l'insieme degli annullabili allora abbiamo sicuramente fatto un errore.

Possiamo adesso definire l'insieme dei terminali con cui cominciano le forme sentenziali derivabili da un'altra forma sentenziale.

Definizione 17. Data una grammatica G e una forma sentenziale α, diciamo che un terminale t è un *inizio* di α se e solo se $\alpha \Rightarrow_G^* t\beta$ per qualche forma sentenziale β. L'insieme degli inizi di un non terminale N è indicato come $\mathsf{first}_G(N)$, con G normalmente omessa quando è chiara dal contesto.

Basta conoscere gli inizi dei non terminali per indurre quelli di una qualsiasi forma sentenziale.

Proposizione 3. Sia $\langle T, N, I, P\rangle$ una grammatica e $\nu : N \mapsto \wp(T)$ una funzione che assegna a ciascun non terminale un insieme di terminali. Sia α una forma sentenziale e definiamo $\mathsf{first}(\alpha, \nu)$ come segue:

$$\begin{aligned}
\mathsf{first}(\varepsilon, \nu) &= \emptyset \\
\mathsf{first}(t, \nu) &= \{t\} \quad \text{per ogni } t \in T \\
\mathsf{first}(n, \nu) &= \nu(n) \quad \text{per ogni } n \in N \\
\mathsf{first}(\alpha_1\alpha_2, \nu) &= \begin{cases} \mathsf{first}(\alpha_1, \nu) & \text{se } \alpha_1 \text{ non è annullabile} \\ \mathsf{first}(\alpha_1, \nu) \cup \mathsf{first}(\alpha_2, \nu) & \text{se } \alpha_1 \text{ è annullabile.} \end{cases}
\end{aligned} \tag{3.7}$$

Se ν assegna a ciascun non terminale l'insieme dei suoi inizi, allora gli inizi di α sono esattamente $\mathsf{first}(\alpha, \nu)$. In tal caso scriviamo $\mathsf{first}(\alpha)$ piuttosto che $\mathsf{first}(\alpha, \nu)$.

Si noti che la definizione dei first richiede la conoscenza dei nullable. In particolare, i nullable sono usati nel calcolo degli inizi di una forma sentenziale del tipo $\alpha_1\alpha_2$: i suoi inizi sono infatti gli inizi di α_1 ma, se α_1 è annullabile, bisogna aggiungere anche gli inizi di α_2.

Anche il calcolo degli insiemi first per una grammatica si effettua come un calcolo di punto fisso. Inizialmente gli inizi di ciascun non terminale sono considerati vuoti (ϕ^0). Quindi si aggiungono agli inizi di ciascun non terminale l'insieme dei terminali con cui inizia il lato destro di una produzione per tale non terminale. Per calcolare come iniziano i lati destri delle produzioni si usa l'approssimazione precedente, quindi all'inizio si usa ϕ^0. Si ottengono così delle approssimazioni successive ϕ^1, ϕ^2,... finché non si raggiunge il punto fisso, cioè un k tale che $\phi^k = \phi^{k+1}$.

Proposizione 4. Sia $\langle T, N, I, P\rangle$ una grammatica e $\nu : N \mapsto \wp(T)$. Definiamo

$$\phi(\nu)(L) = \bigcup_{L \to r \in P} \mathsf{first}(r, \nu),$$

per ogni $L \in N$. Si ha $\mathsf{first}_G = lfp(\phi)$, dove $lfp(\phi)$ è il minimo punto fisso di ϕ, calcolabile come $\lim_{i\to\infty} \phi^i$, con

$$\phi^0(L) = \emptyset$$
$$\phi^{i+1}(L) = \phi(\phi^i)(L) \quad \text{per ogni } i \geq 0$$

e per ogni $L \in N$.

Mostriamo per esempio il calcolo degli inizi dei non terminali della grammatica in Figura 3.6. Useremo una tabella che rappresenta le approssimazioni ϕ^i, indicando per ciascun non terminale l'insieme degli inizi già calcolati. All'inizio tali insiemi sono vuoti:

first

	ϕ^0
I	$\emptyset$
com	$\emptyset$
exp	$\emptyset$

Per calcolare ϕ^1 dobbiamo considerare ciascuna produzione della grammatica. La produzione $I \to com$ \$, dal momento che com non è annullabile (tabella (3.5)), implica che $\phi^1(I) = \mathsf{first}(com\ \$, \phi^0) = \phi^0(com) = \emptyset$. La produzione $com \to exp$ `ASSIGN INTEGER`, dal momento che exp non è annullabile, implica che $\phi^1(com) = \mathsf{first}(exp\ \texttt{ASSIGN INTEGER}, \phi^0) = \phi^0(exp) = \emptyset$. Le tre produzioni $exp \to$ `ID`, $exp \to$ `INTEGER` ed $exp \to$ `MINUS` exp implicano che $\phi^1(exp) = \mathsf{first}(\texttt{ID}, \phi^0) \cup \mathsf{first}(\texttt{INTEGER}, \phi^0) \cup \mathsf{first}(\texttt{MINUS}\ exp, \phi^0) = \{\texttt{ID}, \texttt{INTEGER}, \texttt{MINUS}\}$. In conclusione otteniamo:

first

	ϕ^0	ϕ^1
I	$\emptyset$	$\emptyset$
com	$\emptyset$	$\emptyset$
exp	$\emptyset$	{`ID`, `INTEGER`, `MINUS`}

Dal momento che $\phi^0 \neq \phi^1$, non abbiamo ancora ottenuto il punto fisso e dobbiamo calcolare ϕ^2. Il procedimento è simile a quello per calcolare ϕ^1, con l'unica differenza che adesso $\phi^1(com) = \mathsf{first}(exp\ \texttt{ASSIGN INTEGER}, \phi^1) = \phi^1(exp) = \{\texttt{ID}, \texttt{INTEGER}, \texttt{MINUS}\}$. Otteniamo quindi la tabella:

first

	ϕ^0	ϕ^1	ϕ^2
I	$\emptyset$	$\emptyset$	$\emptyset$
com	$\emptyset$	$\emptyset$	{`ID`, `INTEGER`, `MINUS`}
exp	$\emptyset$	{`ID`, `INTEGER`, `MINUS`}	{`ID`, `INTEGER`, `MINUS`}

Abbiamo ancora $\phi^1 \neq \phi^2$ per cui dobbiamo calcolare ϕ^3. L'unica differenza è che adesso otteniamo $\phi^3(I) = \mathsf{first}(com\ \$, \phi^2) = \phi^2(com) = \{\texttt{ID}, \texttt{INTEGER}, \texttt{MINUS}\}$. In conclusione otteniamo la

tabella:

first

	ϕ^0	ϕ^1	ϕ^2	ϕ^3
I	$\emptyset$	$\emptyset$	$\emptyset$	{ID, INTEGER, MINUS}
com	$\emptyset$	$\emptyset$	{ID, INTEGER, MINUS}	{ID, INTEGER, MINUS}
exp	$\emptyset$	{ID, INTEGER, MINUS}	{ID, INTEGER, MINUS}	{ID, INTEGER, MINUS}

(3.8)

Si può verificare che se calcolassimo ϕ^4 otterremmo $\phi^4 = \phi^3$, per cui abbiamo raggiunto il punto fisso e possiamo dire che gli inizi dei non terminali della grammatica in Figura 3.6 sono sempre {ID, INTEGER, MINUS}.

Anche la funzione ϕ della Proposizione 4 è monotona crescente, per cui gli insiemi di inizi calcolati durante le iterazioni di punto fisso non possono mai decrescere. Se quindi essi decrescessero è perché abbiamo fatto un errore nel calcolo.

L'esempio precedente è relativamente semplice dal momento che nessun non terminale è annullabile nella grammatica in Figura 3.6. Consideriamo invece la grammatica in Figura 3.8, di cui avevamo determinato che i non terminali annullabili sono *A*, *B* ed *L*. Il calcolo degli inizi comincia con l'insieme vuoto di inizi per ciascun non terminale:

first

	ϕ^0
I	$\emptyset$
L	$\emptyset$
A	$\emptyset$
B	$\emptyset$

Per calcolare ϕ^1, consideriamo le produzioni della grammatica. Da $I \to L\$$ concludiamo che

$$\phi^1(I) = \mathsf{first}(L\$, \phi^0) = \mathsf{first}(L, \phi^0) \cup \mathsf{first}(\$, \phi^0) = \emptyset \cup \{\$\} = \{\$\}.$$

Si noti che questo è una conseguenza del fatto che *L* è annullabile, come abbiamo ricordato. Otteniamo inoltre

$$\phi^1(L) = \mathsf{first}(AB, \phi^0) = \mathsf{first}(A, \phi^0) \cup \mathsf{first}(B, \phi^0) = \emptyset \cup \emptyset = \emptyset$$

poiché anche *A* è annullabile. Infine abbiamo

$$\phi^1(A) = \mathsf{first}(\varepsilon, \phi^0) \cup \mathsf{first}(\mathsf{a}A, \phi^0) = \emptyset \cup \mathsf{first}(\mathsf{a}, \phi^0) = \mathsf{first}(\mathsf{a}, \phi^0) = \{\mathsf{a}\}.$$

Si noti che questo è una conseguenza del fatto che a non è annullabile. Similmente otteniamo $\phi^1(B) = \{\mathsf{b}\}$. In conclusione:

first

	ϕ^0	ϕ^1
I	$\emptyset$	{$}
L	$\emptyset$	$\emptyset$
A	$\emptyset$	{a}
B	$\emptyset$	{b}

Dal momento che $\phi^1 \neq \phi^0$ dobbiamo calcolare ϕ^2. L'unica differenza sarà nel calcolo di $\phi^2(L)$, dal momento che adesso, ricordando che A è annullabile, otteniamo

$$\phi^2(L) = \mathsf{first}(AB, \phi^1) = \mathsf{first}(A, \phi^1) \cup \mathsf{first}(B, \phi^1) = \{\mathsf{a}\} \cup \{\mathsf{b}\} = \{\mathsf{a}, \mathsf{b}\}.$$

Otteniamo quindi la tabella

first

	ϕ^0	ϕ^1	ϕ^2
I	$\emptyset$	{$}	{$}
L	$\emptyset$	$\emptyset$	{a, b}
A	$\emptyset$	{a}	{a}
B	$\emptyset$	{b}	{b}

Poiché $\phi^2 \neq \phi^1$ dobbiamo ancora calcolare ϕ^3. L'unica differenza è che adesso otteniamo

$$\phi^3(I) = \mathsf{first}(L\$, \phi^2) = \mathsf{first}(L, \phi^2) \cup \mathsf{first}(\$, \phi^2) = \{\mathsf{a}, \mathsf{b}\} \cup \{\$\} = \{\mathsf{a}, \mathsf{b}, \$\}.$$

Otteniamo quindi la tabella

first

	ϕ^0	ϕ^1	ϕ^2	ϕ^3
I	$\emptyset$	{$}	{$}	{a, b, $}
L	$\emptyset$	$\emptyset$	{a, b}	{a, b}
A	$\emptyset$	{a}	{a}	{a}
B	$\emptyset$	{b}	{b}	{b}

(3.9)

Dobbiamo ancora calcolare ϕ^4, ma si può verificare che $\phi^4 = \phi^3$, per cui ϕ^3 è il punto fisso cercato.

Definiamo adesso l'ultima informazione che vogliamo derivare da una grammatica: i *seguiti* di un non terminale. Come abbiamo già accennato, essi ci serviranno per quelle produzioni il cui lato destro è annullabile, per cui dobbiamo conoscere cosa può seguire il loro lato sinistro per poterle selezionare. Si noti che, a differenza degli annullabili e dei primi che sono definiti per tutte le forme sentenziali, i seguiti sono definiti *solo per i non terminali* della grammatica. Essi sono quei terminali che si possono trovare dopo il non terminale in una derivazione sviluppata dal non terminale iniziale. Conseguentemente possiamo dire, informalmente, che i seguiti di un non terminale X sono quei terminali (token) con cui inizia, nel linguaggio della grammatica, quel che segue una stringa derivabile da X.

Definizione 18. Data una grammatica $\langle T, N, I, P\rangle$ e $X \in N$, diciamo che t è un *seguito* di X se e solo se $I \Rightarrow_G^* \alpha X t \beta$ per qualche forma sentenziale α e β (possibilmente vuote). L'insieme dei seguiti di X è indicato con $\mathsf{follow}_G(X)$, con G normalmente omessa quando è chiara dal contesto.

Il calcolo dei seguiti, ancora una volta come punto fisso, avviene partendo da una approssimazione iniziale ϕ^0 in cui i seguiti di ciascun non terminale sono l'insieme vuoto. Quindi si calcola ϕ^1 considerando tutte le occorrenze di un non terminale X alla destra delle produzioni. Per ogni occorrenza si aggiungono ai seguiti di X gli inizi di ciò che segue X nel lato destro della produzione e, se quel che segue X è annullabile, anche l'approssimazione corrente per i seguiti del lato sinistro della produzione, fornita da ϕ^0. Questo procedimento è iterato fino al punto fisso, cioè a quel k per cui si ha $\phi^k = \phi^{k+1}$.

Proposizione 5. Sia $\langle T, N, I, P\rangle$ una grammatica e $\nu : N \mapsto \wp(T)$. Definiamo

$$\phi(\nu)(X) = \bigcup_{L \to \alpha X \beta \in P} \begin{cases} \mathsf{first}(\beta) & \text{se } \beta \text{ non è annullabile} \\ \mathsf{first}(\beta) \cup \nu(L) & \text{se } \beta \text{ è annullabile} \end{cases}$$

per ogni $X \in N$. Si ha[3] $\mathsf{follow}_G = lfp(\phi)$, dove $lfp(\phi)$ è il minimo punto fisso di ϕ, calcolabile come $\lim_{i\to\infty} \phi^i$, con

$$\begin{aligned} \phi^0(X) &= \emptyset \\ \phi^{i+1}(X) &= \phi(\phi^i)(X) \quad \text{per ogni } i \geq 0 \end{aligned}$$

e per ogni $X \in N$.

Si noti che il calcolo dei seguiti richiede la conoscenza degli annullabili e dei primi, per cui deve essere effettuato dopo il calcolo di questi ultimi.

Consideriamo per esempio il calcolo dei seguiti per la grammatica in Figura 3.6. Ricordiamo che abbiamo già determinato che nessun non terminale è annullabile in tale grammatica e che gli inizi dei non terminali sono costantemente {`ID`, `INTEGER`, `MINUS`} (tabelle (3.5) e (3.8)). Partiamo dall'approssimazione iniziale per i seguiti:

follow

	ϕ^0
I	$\emptyset$
com	$\emptyset$
exp	$\emptyset$

Per calcolare ϕ^1 consideriamo, per ciascun non terminale, dove esso occorre alla destra di una produzione della grammatica. Il non terminale I non occorre mai alla destra di una produzione.

[3] Questo risultato è vero sotto alcune ipotesi sulla grammatica, normalmente vere, come per esempio che tutte le sue regole siano raggiungibili in una derivazione da I. In caso contrario quel che si ottiene è una approssimazione per eccesso dell'insieme dei seguiti, che comunque va benissimo per tutti i nostri scopi. Non entriamo qui in questi dettagli.

Conseguentemente avremo $\phi^1(I) = \emptyset$. Il non terminale *com* occorre alla destra della produzione $I \to com\ \$$. In questo caso abbiamo $\alpha = \varepsilon$ e $\beta = \$$. Dal momento che β non è annullabile (è un terminale), otteniamo $\phi^1(com) = \mathsf{first}(\$) = \{\$\}$. Il non terminale *exp* occorre alla destra della produzione $com \to exp$ `ASSIGN INTEGER`. In questo caso abbiamo $\alpha = \varepsilon$ e $\beta =$ `ASSIGN INTEGER`, che non è annullabile. Il non terminale *exp* occorre anche alla destra della produzione $exp \to$ `MINUS` *exp*. In questo altro caso abbiamo $\alpha =$ `MINUS` e $\beta = \varepsilon$ che è chiaramente annullabile. Abbiamo quindi $\phi^1(exp) = \mathsf{first}(\texttt{ASSIGN INTEGER}) \cup \mathsf{first}(\varepsilon) \cup \phi^0(exp) = \{\texttt{ASSIGN}\} \cup \emptyset \cup \emptyset = \{\texttt{ASSIGN}\}$. Il risultato è quindi

follow

	ϕ^0	ϕ^1
I	$\emptyset$	$\emptyset$
com	$\emptyset$	$\{\$\}$
exp	$\emptyset$	{ASSIGN}

Per calcolare ϕ^2 reiteriamo il calcolo a partire da ϕ^1. L'unica differenza è che adesso abbiamo $\phi^2(exp) = \mathsf{first}(\texttt{ASSIGN INTEGER}) \cup \mathsf{first}(\varepsilon) \cup \phi^1(exp) = \{\texttt{ASSIGN}\} \cup \emptyset \cup \{\texttt{ASSIGN}\} = \{\texttt{ASSIGN}\}$. Il risultato quindi non cambia:

follow

	ϕ^0	ϕ^1	ϕ^2
I	$\emptyset$	$\emptyset$	$\emptyset$
com	$\emptyset$	$\{\$\}$	$\{\$\}$
exp	$\emptyset$	{ASSIGN}	{ASSIGN}

(3.10)

Concludiamo che $\phi^1 = \phi^2$ è il punto fisso cercato.

Ancora una volta osserviamo che gli insiemi dei seguiti non possono decrescere durante il calcolo del punto fisso. Inoltre se un non terminale non occorre mai alla destra di una produzione, i suoi seguiti saranno costantemente $\emptyset$ durante il calcolo. Questo è il caso di *I* nella tabella precedente (si osservi la grammatica in Figura 3.6). Infine, se un non terminale *N*, quando occorre alla destra di una produzione, è sempre e solo seguito da terminali, allora tali terminali sono, costantemente a partire da ϕ^1, l'insieme dei seguiti di *N*. Questo è il caso di *com* nella tabella precedente.

Consideriamo adesso la grammatica in Figura 3.8. Abbiamo già determinato che l'insieme dei non terminali annullabili è $\{L, A, B\}$ e che gli inizi di ciascun non terminale sono come nella colonna ϕ^3 della tabella (3.9). La prima approssimazione ϕ^0 è

follow

	ϕ^0
I	$\emptyset$
L	$\emptyset$
A	$\emptyset$
B	$\emptyset$

Per calcolare ϕ^1, consideriamo dove ciascun non terminale occorre alla destra delle produzioni della grammatica. Il non terminale I non occorre mai alla destra di una produzione della grammatica, per cui si avrà $\phi^1(I) = \emptyset$ (e così anche per le iterazioni successive). Il non terminale L occorre solo alla destra della produzione $I \to L\$$ per cui avremo $\alpha = \varepsilon$, $\beta = \$$, che non è annullabile, e $\phi^1(L) = \mathsf{first}(\beta) = \{\$\}$ (e così anche per le iterazioni successive). Il non terminale A occorre sia alla destra della produzione $L \to AB$ che alla destra della produzione $A \to \mathsf{a}A$. Nel primo caso si ha $\alpha = \varepsilon$ e $\beta = B$, che è annullabile, e nel secondo caso si ha $\alpha = \mathsf{a}$ e $\beta = \varepsilon$, chiaramente annullabile. Conseguentemente otteniamo $\phi^1(A) = \mathsf{first}(B) \cup \phi^0(L) \cup \mathsf{first}(\varepsilon) \cup \phi^0(A) = \{\mathsf{b}\} \cup \emptyset \cup \emptyset \cup \emptyset = \{\mathsf{b}\}$. Il non terminale B occorre sia alla destra della produzione $L \to AB$ che alla destra della produzione $B \to \mathsf{b}B$. Nel primo caso si ha $\alpha = A$ e $\beta = \varepsilon$, chiaramente annullabile, e nel secondo caso si ha $\alpha = \mathsf{b}$ e $\beta = \varepsilon$, chiaramente annullabile. Concludiamo che $\phi^1(B) = \mathsf{first}(\varepsilon) \cup \phi^0(L) \cup \mathsf{first}(\varepsilon) \cup \phi^0(B) = \emptyset \cup \emptyset \cup \emptyset \cup \emptyset = \emptyset$. Il risultato è quindi

follow

	ϕ^0	ϕ^1
I	$\emptyset$	$\emptyset$
L	$\emptyset$	$\{\$\}$
A	$\emptyset$	$\{\mathsf{b}\}$
B	$\emptyset$	$\emptyset$

Nel calcolo di ϕ^2 da ϕ^1 l'unica differenza è che adesso $\phi^2(A) = \mathsf{first}(B) \cup \phi^1(L) \cup \mathsf{first}(\varepsilon) \cup \phi^1(A) = \{\mathsf{b}\} \cup \{\$\} \cup \emptyset \cup \{\mathsf{b}\} = \{\mathsf{b}, \$\}$ e che $\phi^2(B) = \mathsf{first}(\varepsilon) \cup \phi^1(L) \cup \mathsf{first}(\varepsilon) \cup \phi^1(B) = \emptyset \cup \{\$\} \cup \emptyset \cup \emptyset = \{\$\}$. Otteniamo quindi la tabella:

follow

	ϕ^0	ϕ^1	ϕ^2
I	$\emptyset$	$\emptyset$	$\emptyset$
L	$\emptyset$	$\{\$\}$	$\{\$\}$
A	$\emptyset$	$\{\mathsf{b}\}$	$\{\mathsf{b}, \$\}$
B	$\emptyset$	$\emptyset$	$\{\$\}$

Nel calcolo di ϕ^3 da ϕ^2 l'unica differenza è che adesso $\phi^3(A) = \mathsf{first}(B) \cup \phi^2(L) \cup \mathsf{first}(\varepsilon) \cup \phi^2(A) = \{\mathsf{b}\} \cup \{\$\} \cup \emptyset \cup \{\mathsf{b}, \$\} = \{\mathsf{b}, \$\}$ e inoltre $\phi^3(B) = \mathsf{first}(\varepsilon) \cup \phi^2(L) \cup \mathsf{first}(\varepsilon) \cup \phi^2(B) = \emptyset \cup \{\$\} \cup \emptyset \cup \{\$\} = \{\$\}$. Otteniamo in conclusione la tabella:

follow

	ϕ^0	ϕ^1	ϕ^2	ϕ^3
I	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
L	$\emptyset$	$\{\$\}$	$\{\$\}$	$\{\$\}$
A	$\emptyset$	$\{\mathsf{b}\}$	$\{\mathsf{b}, \$\}$	$\{\mathsf{b}, \$\}$
B	$\emptyset$	$\emptyset$	$\{\$\}$	$\{\$\}$

(3.11)

e concludiamo che $\phi^2 = \phi^3$ è il punto fisso cercato.

3.3.2 La tabella *LL*(1) e la costruzione del parser *LL*(1)

Una volta calcolati gli insiemi nullable, first e follow per una grammatica G, siamo nelle condizioni di scrivere un programma, come quello in Figura 3.7, che riconosce tutte e sole le stringhe del linguaggio generato da G. Basta determinare, per ogni produzione, l'insieme dei *terminali discriminanti*, ovvero capaci di guidare il parser verso l'applicazione della giusta produzione sulla base del token che ci sta davanti, detto *lookahead*.

Definizione 19. Data una grammatica G è una sua produzione $L \to r$, l'insieme dei *terminali discriminanti* per $L \to r$ è dato da $\mathsf{first}(r)$ se r non è annullabile e da $\mathsf{first}(r) \cup \mathsf{follow}(L)$ se r è annullabile.

Riportiamo ad esempio i terminali discriminanti per la grammatica in Figura 3.6, costruiti grazie alle tabelle (3.5), (3.8) e (3.10):

$$\begin{aligned} I &\to com\ \$ && \{\texttt{ID}, \texttt{INTEGER}, \texttt{MINUS}\} \\ com &\to exp\ \texttt{ASSIGN}\ \texttt{INTEGER} && \{\texttt{ID}, \texttt{INTEGER}, \texttt{MINUS}\} \\ exp &\to \texttt{ID} && \{\texttt{ID}\} \\ exp &\to \texttt{INTEGER} && \{\texttt{INTEGER}\} \\ exp &\to \texttt{MINUS}\ exp && \{\texttt{MINUS}\} \end{aligned}$$

Il parser Java può quindi essere scritto usando gli insiemi discriminanti per decidere quale produzione applicare per quei non terminali che sono definiti da più di una produzione (come *exp* nell'esempio sopra). Si guardi per esempio il comando `switch` in Figura 3.7.

Questo metodo di parsing è detto *LL*(1) o *parsing a discesa ricorsiva con lookahead unitario*. Il motivo del nome *LL*(1) è che il parser risultante legge il file sorgente da sinistra a destra (da cui la prima *L* da *left-to-right*) e genera derivazioni leftmost (da cui la seconda *L*); inoltre esso usa un solo carattere di lookahead per decidere quale produzione applicare per i non terminali definiti da più di una produzione.

È tradizione indicare in maniera compatta un parser *LL*(1) tramite una *tabella LL*(1) che ha sulle ascisse i terminali e sulle ordinate i non terminali della grammatica. All'incrocio della riga etichettata come L con la colonna etichettata con t si mette la produzione $L \to r$ della grammatica che deve essere usata per L di fronte al non terminale t. Sulla base degli insiemi discriminanti calcolati sopra, otteniamo la seguente tabella *LL*(1) per la grammatica in Figura 3.6:

	\$	ASSIGN	ID	INTEGER	MINUS
I			$I \to com\ \$$	$I \to com\ \$$	$I \to com\ \$$
com			$com \to exp$ ASSIGN INTEGER	$com \to exp$ ASSIGN INTEGER	$com \to exp$ ASSIGN INTEGER
exp			$exp \to$ ID	$exp \to$ INTEGER	$exp \to$ MINUS *exp*

(3.12)

Le caselle vuote sono in realtà condizioni di errore di sintassi. Se per esempio dobbiamo identificare una *exp* e davanti a noi c'è il token \$ o ASSIGN allora il file sorgente non appartiene al linguaggio generato dalla grammatica (si veda la Figura 3.7).

Se una tabella *LL*(1) contiene al più una produzione per casella allora è possibile scrivere un parser Java come quello in Figura 3.7.

$$
\begin{aligned}
I &\to A\$ \\
A &\to \mathtt{a} \\
A &\to A\mathtt{a}
\end{aligned}
$$

Figura 3.9: Una grammatica non *LL*(1) ma *LR*(0).

Definizione 20. Una grammatica è *LL*(1) se e solo se la sua tabella *LL*(1) non presenta *conflitti*, ovvero caselle che contengono più di una produzione. In maniera equivalente, una grammatica è *LL*(1) se non ha due produzioni con lo stesso lato sinistro e con insiemi discriminanti non disgiunti. Un linguaggio è *LL*(1) se e solo se ammette una grammatica *LL*(1).

Per esempio la grammatica in Figura 3.6 è *LL*(1) poiché la tabella (3.12) non presenta conflitti.

Se consideriamo la grammatica (3.8) e calcoliamo i suoi insiemi discriminanti, otteniamo (si consultino le tabelle (3.6), (3.9) e (3.11)):

$$
\begin{aligned}
I &\to L\$ && \{\mathtt{a}, \mathtt{b}, \$\} \\
L &\to AB && \{\mathtt{a}, \mathtt{b}, \$\} \\
A &\to \varepsilon && \{\mathtt{b}, \$\} \\
A &\to \mathtt{a}A && \{\mathtt{a}\} \\
B &\to \varepsilon && \{\$\} \\
B &\to \mathtt{b}B && \{\mathtt{b}\}
\end{aligned}
$$

Già dagli insiemi discriminanti si comprende che la grammatica è *LL*(1). Se costruiamo la sua tabella *LL*(1) otteniamo:

	$	a	b
I	$I \to L\$$	$I \to L\$$	$I \to L\$$
L	$L \to AB$	$L \to AB$	$L \to AB$
A	$A \to \varepsilon$	$A \to \mathtt{a}A$	$A \to \varepsilon$
B	$B \to \varepsilon$		$B \to \mathtt{b}B$

(3.13)

Anche questa volta la tabella non contiene conflitti, per cui la grammatica in Figura 3.8 è *LL*(1).

Esercizio 13. Si usi la tabella (3.13) per scrivere il programma Java che implementa un parser *LL*(1) per la grammatica in Figura 3.8. Si faccia attenzione al codice per le ε-produzioni!

Il parsing *LL*(1) è molto intuitivo e semplice da implementare. Purtroppo è anche poco potente. Per esempio, se consideriamo la grammatica in Figura 3.9 otteniamo:

nullable

	ϕ^0
I	*false*
A	*false*

first

	ϕ^0	ϕ^1
I	$\emptyset$	{a}
A	$\emptyset$	{a}

follow

	ϕ^0	ϕ^1
I	$\emptyset$	$\emptyset$
A	$\emptyset$	{$, a}

Conseguentemente gli insiemi discriminanti sono:

$$\begin{array}{ll} I \to A\$ & \{\mathtt{a}\} \\ A \to \mathtt{a} & \{\mathtt{a}\} \\ A \to A\mathtt{a} & \{\mathtt{a}\} \end{array}$$

e la tabella $LL(1)$ presenta un conflitto:

	\$	a
I		$I \to A\$$
A		$A \to \mathtt{a}$ $A \to A\mathtt{a}$

Intuitivamente, se vogliamo riconoscere il non terminale A e davanti a noi c'è il non terminale a, non riusciamo a scegliere fra le due produzioni $A \to \mathtt{a}$ e $A \to A\mathtt{a}$, poiché entrambe derivano stringhe che cominciano con a. Concludiamo che non esiste un parser $LL(1)$ per tale grammatica.

Esercizio 14. Si dimostri che se una grammatica G contiene due produzioni del tipo $L \to t$ e $L \to L\alpha$, con t terminale ed α forma sentenziale qualsiasi (possibilmente vuota) allora G non è $LL(1)$. Questo implica automaticamente che la grammatica in Figura 3.9 non è $LL(1)$, come abbiamo del resto appena verificato.

Esercizio 15. Si dimostri che se una grammatica G contiene una produzione del tipo $L \to t$ con t terminale e due produzioni del tipo $X \to L\alpha$ e $X \to L\beta$ con α e β forme sentenziali qualsiasi (possibilmente vuote) allora G non è $LL(1)$.

I due precedenti esercizi danno un'idea di quanto poco potenti siano le grammatiche $LL(1)$, dal momento che qualsiasi grammatica per un linguaggio di programmazione *normale* contiene produzioni come quelle considerate in tali esercizi. Basta per esempio guardare le produzioni della Sezione 3.2.3. Ciò nonostante, il parsing $LL(1)$ è così semplice che è stato lungamente utilizzato per scrivere i primi compilatori. Questo spiega perché *vecchi* linguaggi come il LISP abbiano una sintassi *scomoda*, come per esempio una notazione prefissa per gli operatori aritmetici, finalizzata ad eliminare i conflitti del parsing $LL(1)$.

La teoria del parsing $LL(1)$ può essere generalizzata al parsing $LL(k)$, in cui si utilizzano fino a k caratteri davanti al punto di programma in cui ci troviamo per determinare quale produzione applicare per lo stesso non terminale. I problemi degli esercizi 14 e 15 si riducono con l'aumentare di k ma non scompaiono del tutto. La sezione seguente descrive invece una tecnica nettamente più potente per effettuare il parsing.

Esercizio 16. Come devono essere fatte le produzioni di una grammatica $LL(0)$? Riuscite a definire un linguaggio che ammette una grammatica $LL(0)$?

3.4 Il parsing *LR*

In questa sezione descriviamo un'altra tecnica di parsing, detta *LR*, meno intuitiva di quella della Sezione 3.3 e meno semplice da implementare. Essa però è più potente del parsing *LL*. Una versione del parsing *LR* è utilizzata da JavaCup per generare il parser di Kitten a partire dalla grammatica che abbiamo descritto nella Sezione 3.2. Nelle prossime sezioni consideriamo ciascuna delle varie versioni di parsing *LR*.

3.4.1 Il parsing *LR*(0)

Riconsideriamo la grammatica in Figura 3.9. Abbiamo già visto che essa non è una grammatica *LL*(1). Ragioniamo su come deve cominciare una derivazione di una stringa che appartiene al linguaggio generato da tale grammatica. Sicuramente il primo passo sarà l'utilizzo dell'unica produzione per il non terminale iniziale I, cioè la derivazione deve iniziare con $I \Rightarrow A\$$. Questo significa che, all'inizio del parsing, ci aspettiamo che davanti a noi ci sia una stringa derivabile da A seguita dal carattere \$ di fine file. Scriviamo questa *previsione* come

$$I \to .\, A\$$$

che indica, alla sinistra del punto, la parte già trovata nel file sorgente del lato destro della produzione (in questo caso ε) e, alla destra del punto, la parte che ancora ci aspettiamo di trovare (in questo caso $A\$$). Una produzione della grammatica con un punto da qualche parte nel suo lato destro è detta *item LR*(0) della grammatica. Un item *LR*(0) indica quindi che siamo in uno *stato* in cui ci aspettiamo di potere utilizzare la produzione purché davanti a noi ci sia qualcosa derivabile da quel che segue il punto nell'item. Si osservi che se, come in $I \to .\, A\$$, ci aspettiamo che davanti a noi ci sia una A, allora è possibile utilizzare la produzione $A \to \mathtt{a}$ o la produzione $A \to A\mathtt{a}$ per derivare tale A. Per cui all'inizio del parsing l'insieme delle produzioni che ci aspettiamo di poter utilizzare è

$$\boxed{\begin{array}{l} I \to .\, A\$ \\ A \to .\, \mathtt{a} \\ A \to .\, A\mathtt{a} \end{array}} \tag{3.14}$$

Un insieme di item come quello sopra è detto *stato LR*(0). Si noti che uno stato *LR*(0) deve essere *chiuso*, ovvero deve contenere tutte le produzioni per i non terminali che stanno immediatamente alla destra di un punto.

Definizione 21. Sia $G = \langle T, N, I, P \rangle$ una grammatica. Un insieme S di item *LR*(0) è *chiuso rispetto a* G se, per ogni $L \to \alpha.\, R\beta \in S$ ed ogni $R \to \gamma \in P$, si ha $R \to .\, \gamma \in S$.

Supponiamo adesso di trovarci nello stato (3.14) e che davanti a noi, nel file sorgente, ci sia qualcosa a cui si può ridurre il non terminale A. Ci sono due item che si aspettano di trovarsi davanti una A: l'item $I \to .\, A\$$ e l'item $I \to .\, A\mathtt{a}$, ovvero quegli item che hanno la A subito dopo il punto. Se quindi riduciamo quello che sta davanti a noi ad una A, finiamo nello stato

$$\boxed{\begin{array}{l} I \to A.\, \$ \\ A \to A.\, \mathtt{a} \end{array}} \tag{3.15}$$

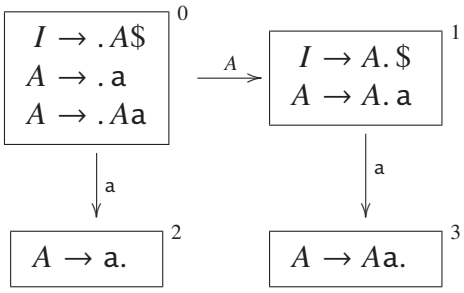


Figura 3.10: L'automa *LR*(0) per la grammatica in Figura 3.9.

Lo stato (3.15) è stato ottenuto dallo stato (3.14) spostando avanti i punti che stanno immediatamente alla sinistra di una A e quindi chiudendo l'insieme di item risultante (che in effetti in questo esempio era già chiuso). Questo stato indica che se davanti a noi ci sarà un carattere \$ allora ridurremo tutto il file sorgente ad una I con la produzione $I \rightarrow A\$$, ovvero dichiareremo che il file sorgente soddisfa le regole della grammatica in Figura 3.9. Questa sarà la condizione di *accettazione* del file sorgente. Se invece ci sarà il carattere a allora ridurremo la A e il carattere a ad una A tramite la produzione $A \rightarrow A\mathsf{a}$. Indichiamo quest'ultima situazione dicendo che se nello stato (3.15) ci troviamo davanti ad una a, allora finiamo nello stato

$$\boxed{A \rightarrow A\mathsf{a}.} \tag{3.16}$$

Lo stato (3.16) è stato ottenuto dallo stato (3.15) spostando avanti il punto che sta immediatamente alla sinistra di una a e quindi chiudendo l'insieme di item risultante (che anche in questo caso era già chiuso). Nello stato (3.16) notiamo che c'è un punto alla fine di un item. Esso indica che abbiamo già visto tutto quello che sta alla sua sinistra e che non rimane nulla ancora da vedere: possiamo quindi dire che gli ultimi caratteri letti dal file sorgente hanno la struttura $A\mathsf{a}$ del lato destro della produzione da cui l'item è derivato e che quindi essi formano una A per via della produzione $A \rightarrow A\mathsf{a}$.

C'è ancora da considerare cosa accade quando, nello stato (3.14), ci troviamo davanti il carattere a. In tal caso finiamo nello stato

$$\boxed{A \rightarrow \mathsf{a}.} \tag{3.17}$$

ottenuto dallo stato (3.14) spostando avanti il punto che sta subito alla sinistra del carattere a e chiudendo poi l'insieme di item risultante (anche in questo caso esso era già chiuso). Lo stato (3.17) indica che abbiamo letto una a dal file sorgente e che essa può essere vista come una A per via della produzione $A \rightarrow \mathsf{a}$.

La Figura 3.10 raccoglie i quattro stati che abbiamo visto sopra, legandoli con delle transizioni che indicano la condizione sotto la quale si passa da uno stato all'altro. Essi sono stati numerati come 0, 1, 2 e 3, ma qualsiasi altra numerazione andrebbe bene, purché lo stato iniziale rimanga numerato come 0. Si noti che una transizione da uno stato s_0 a uno stato s_1 etichettata con un terminale, come a, indica che da s_0, se il prossimo carattere letto dal file sorgente è a,

si passa nello stato s_1; se la transizione fosse invece etichettata con un non terminale A, essa indicherebbe che da s_0 si passa ad s_1 se gli ultimi caratteri *già letti* dal file sorgente formano uno stringa derivabile da A.

Lo schema in Figura 3.10 può essere usato come un *programma* per un *automa a pila* (deterministico), cioè una macchina a stati che, al posto di un singolo stato, dispone di uno stack (*pila*) di stati. Lo stack fornisce all'automa una memoria non limitata a priori, per cui ci aspettiamo che questo tipo di automa sia nettamente più potente di quelli considerati nel Capitolo 2. Inizialmente questo automa ha uno stack di stati che contiene solo lo stato 0 e davanti alla sua *testina di lettura* si trova l'inizio del file sorgente da analizzare. Supponiamo che tale file sia la stringa `aa`$ e vediamo come si comporta l'automa per concludere che tale stringa appartiene al linguaggio della grammatica in Figura 3.9. Rappresentiamo la configurazione iniziale dell'automa come segue:

$$0 \qquad\qquad \texttt{aa\$}$$

A sinistra rappresentiamo lo stack di stati dell'automa (lo stack cresce verso destra) e a destra la stringa del file sorgente che deve ancora venire letta. Dal momento che il prossimo carattere da leggere è la prima `a`, seguendo la Figura 3.10 concludiamo che l'automa va nello stato 2. La testina di lettura si sposta avanti di un carattere:

$$0, 2 \qquad\qquad \texttt{a\$}$$

La Figura 3.10 mostra che lo stato 2 contiene un item con il punto alla fine. Questo significa che abbiamo visto tutti i caratteri del lato destro della produzione e possiamo *ridurli* al lato sinistro della produzione, in questo caso A. Per effettuare questa riduzione, l'automa elimina tanti stati dalla cima dello stack, quanto è lungo il lato destro della produzione. Dal momento che la produzione $A \rightarrow \texttt{a}$ ha un lato destro di lunghezza unitaria (un terminale), dobbiamo eliminare uno stato dallo stack, esponendo lo stato 0 in cima allo stack. Poiché abbiamo appena ridotto qualcosa ad una A, la Figura 3.10 ci dice che l'automa dallo stato 0 va nello stato 1:

$$0, 1 \qquad\qquad \texttt{a\$}$$

Adesso siamo nello stato 1 e davanti a noi c'è una `a`. Secondo la Figura 3.10, l'automa va nello stato 3 leggendo tale `a`:

$$0, 1, 3 \qquad\qquad \$$$

Lo stato 3 contiene un item col punto alla fine. Questo significa che abbiamo già visto qualcosa che ha la struttura del lato destro della produzione da cui l'item è derivato, cioè di $A\texttt{a}$, e che possiamo ridurlo ad una A. Dal momento che $A\texttt{a}$ ha lunghezza due (un non terminale seguito da un terminale), dobbiamo estrarre due stati dallo stack, esponendo lo stato 0 in cima allo stack. Poiché abbiamo appena ridotto qualcosa a una A, la Figura 3.10 ci dice che il nostro automa deve andare nello stato 1:

$$0, 1 \qquad\qquad \$$$

Siamo adesso nello stato 1 e davanti a noi c'è il carattere \$ di fine file. Abbiamo detto che questa è la condizione di accettazione, per cui l'automa si ferma *accettando* il file sorgente.

Ecco le produzioni che sono state usate dall'automa che ha accettato la stringa aa\$:

$$A \to \mathtt{a}$$
$$A \to A\mathtt{a}$$

Ordiniamole in senso inverso e aggiungiamo la produzione $I \to A\$$ che è implicitamente usata al momento dell'accettazione:

$$I \to A\$$$
$$A \to A\mathtt{a}$$
$$A \to \mathtt{a}$$

Mettiamole una dopo l'altra a formare una derivazione:

$$I \Rightarrow A\$ \Rightarrow A\mathtt{a}\$ \Rightarrow \mathtt{aa}\$.$$

Questa è proprio la derivazione della stringa aa\$ a partire dal non terminale iniziale I. Si può dimostrare che questa derivazione costruita dall'automa è sempre una derivazione rightmost.

Vediamo cosa accade se invece proviamo ad eseguire l'automa su un file sorgente che contiene la stringa ab\$, che non appartiene al linguaggio della grammatica in Figura 3.9. Le prime transizioni sono simili a quelle viste prima:

0	ab\$
0, 2	b\$
0, 1	b\$

a questo punto però l'automa si trova nello stato 1 e davanti alla testina di lettura c'è una b. Nessuna freccia uscente dallo stato 1 in Figura 3.10 è etichettata con b, per cui l'automa si ferma *rifiutando* il file sorgente.

Ricapitoliamo quindi le operazioni che un automa a pila è capace di compiere:

spostamento di un token: se siamo in uno stato i, sotto la testina di lettura c'è il token t e c'è una freccia da i ad j etichettata con t, allora l'automa *sposta* il token t (cioè avanza la testina di lettura di una posizione nel file sorgente) e aggiunge lo stato j in cima al suo stack di stati;

riduzione secondo una produzione: se siamo in uno stato i che contiene un item $L \to \alpha.$, l'automa elimina dalla cima dello stack tanti stati quanta è la lunghezza l (numero di terminali e non terminali) di α. Se lo stack fosse più corto di $l + 1$ si dà errore. Quindi si prende lo stato k che è stato esposto in cima allo stack e si cerca una transizione uscente da k etichettata con L. Se tale transizione porta nello stato j si aggiunge j in cima allo stack. Se essa non esiste si dà errore;

accettazione: se siamo in uno stato i che contiene una produzione $L \to \alpha.\,\$$ accettiamo il file sorgente;

errore: in tutti gli altri casi si dà errore, rifiutando quindi il file sorgente.

Si noti che le operazioni precedenti devono essere mutuamente esclusive o altrimenti l'automa diventerebbe non deterministico.

Abbiamo detto che lo schema in Figura 3.10 è una sorta di *programma* per l'automa a pila. Esso guida l'esecuzione dell'automa. Per questo motivo tale schema viene chiamato *automa LR*(0). È conveniente e compatto rappresentare tale automa tramite una *tabella LR*(0), che indica cosa fare in ogni stato sulla base del carattere che sta sotto la testina di lettura dell'automa. Sarà questa tabella e non lo schema che verrà effettivamente inserita nell'automa per programmarlo a riconoscere una data grammatica.

Definizione 22. Sia G una grammatica e sia dato il suo automa *LR*(0). La *tabella LR*(0) per G è una tabella che ha sulle ordinate gli stati dell'automa, sulle ascisse i terminali e i non terminali di G e tale che

- per ogni freccia da uno stato i a uno stato j etichettata con un terminale t la casella (i,t) della tabella contiene sj (*sposta e vai in* j);
- per ogni stato i che contiene un item $L \to \alpha.$, dove $L \to \alpha$ è la k-esima produzione della grammatica, la parte dei terminali della riga i-esima della tabella contiene rk (*riduci secondo la produzione* k);
- per ogni freccia da uno stato i a uno stato j etichettato con un non terminale N la casella (i,N) della tabella contiene gj (*vai in* j);
- per ogni stato i che contiene un item del tipo $L \to \alpha.\,\$$, la casella $(i,\$)$ contiene a (*accetta*).

Per esempio la tabella *LR*(0) per la grammatica in Figura 3.9, costruibile grazie allo schema in Figura 3.10, è

	$	a	A	I
0		$s2$	$g1$	
1	a	$s3$		
2	$r1$	$r1$		
3	$r2$	$r2$		

(3.18)

Abbiamo numerato le produzioni della grammatica da 0 (in alto) a 3 (in basso). Le caselle vuote della tabella vanno interpretate come delle situazioni di errore.

Definizione 23. Una grammatica è *LR*(0) se la sua tabella *LR*(0) non contiene *conflitti*, cioè caselle con più di un contenuto. Un linguaggio è *LR*(0) se ha una grammatica *LR*(0).

Dal momento che la tabella (3.18) non ha conflitti, concludiamo che la grammatica in Figura 3.9 è *LR*(0). Ricordiamo che essa invece non è una grammatica *LL*(1).

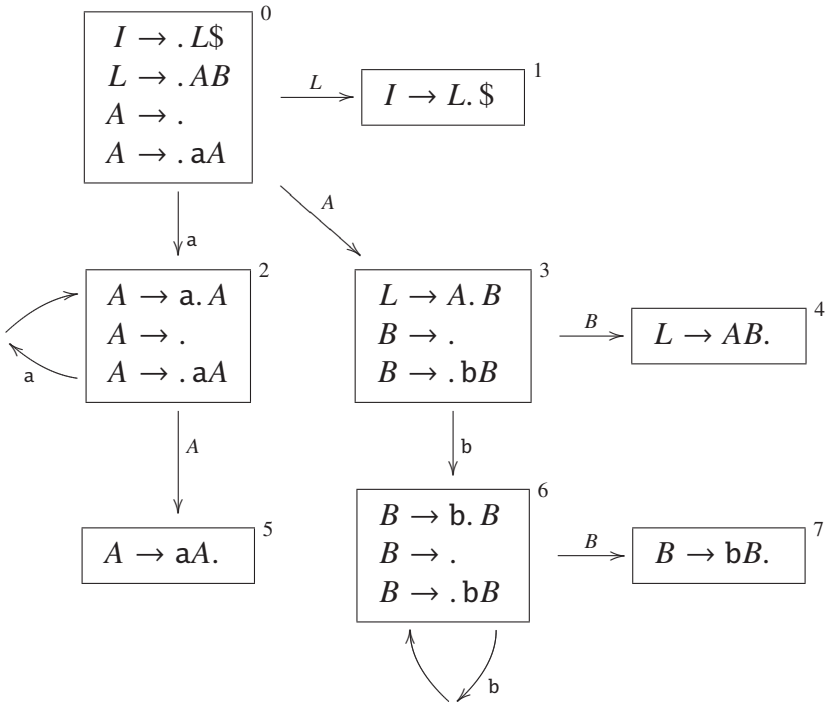


Figura 3.11: L'automa *LR*(0) per la grammatica in Figura 3.8.

Questo metodo di parsing è detto *LR*(0) poiché il parser legge il file sorgente da sinistra a destra (da cui la *L* da *left-to-right*) e genera derivazioni rightmost (da cui la *R*); inoltre esso non usa alcun carattere di lookahead per decidere secondo quale produzione ridurre in uno stato che ha un item con un punto alla fine. Questo è evidente dalla Definizione 22, che dice di inserire le riduzioni per *tutti* i terminali di una riga. Si noti che questo non vuol dire che un parser *LR*(0) non usa alcun lookahead: in effetti esso usa un lookahead unitario ma solo per distinguere azioni che non sono riduzioni (si veda per esempio la riga 1 della tabella (3.18)).

Un grande vantaggio del parsing *LR*(0) rispetto a quello *LL*(1) e a quelli che considereremo nelle prossime sezioni è che non serve calcolare gli insiemi nullable, first e follow della Sezione 3.3.1. Il fatto che la grammatica 3.9 sia *LR*(0) ma non *LL*(1) non deve però indurre a facili entusiasmi, come per esempio a pensare che *LR*(0) sia sempre più potente di *LL*(1). In effetti si può dimostrare che *LR*(0) è strettamente più potente di *LL*(0) (ovvero, ogni grammatica *LL*(0) è anche *LR*(0)) ma è facile trovare una grammatica *LL*(1) che non è *LR*(0). Questo è il caso della grammatica in Figura 3.8, di cui adesso costruiamo automa e tabella *LR*(0), mostrando che quest'ultima contiene dei conflitti.

L'automa *LR*(0) per la grammatica in Figura 3.8 è mostrato in Figura 3.11. Si noti che nello stato 0, partendo dall'item iniziale $I \to .L\$$, abbiamo aggiunto, per chiusura, l'item $L \to .AB$ per la produzione per *L* e quindi, poiché il punto è adesso davanti al non terminale *A*, anche gli item per *A*. L'item $A \to .$ è derivato dalla produzione $A \to \varepsilon$. Si noti inoltre che se nello stato 2 ci troviamo davanti a una a, l'item $A \to .\,\mathtt{a}A$ sposta il punto ottenendo l'item $A \to \mathtt{a}.A$ che, per

chiusura, genera gli item $A \to .$ e $A \to .\,\mathtt{a}A$. Conseguentemente c'è una freccia dallo stato 2 allo stesso stato 2 etichettata con a. Simile il ragionamento per lo stato 6.

Numerando le produzioni in Figura 3.8 da 0 (in alto) a 7 (in basso), la tabella *LR*(0) per tale grammatica è

	$	a	b	*I*	*L*	*A*	*B*
0	*r*2	*s*2/*r*2	*r*2		*g*1	*g*3	
1	*a*						
2	*r*2	*s*2/*r*2	*r*2			*g*5	
3	*r*4	*r*4	*s*6/*r*4				*g*4
4	*r*1	*r*1	*r*1				
5	*r*3	*r*3	*r*3				
6	*r*4	*r*4	*s*6/*r*4				*g*7
7	*r*5	*r*5	*r*5				

(3.19)

Questa volta la tabella contiene molti conflitti sposta/riduci per cui la grammatica non è *LR*(0).

3.4.2 Il parsing *SLR*

Nella sezione precedente abbiamo visto che la grammatica in Figura 3.8 non è *LR*(0). Il motivo è che la sua tabella *LR*(0) contiene dei conflitti, causati dall'aver messo le riduzioni su *tutti* i terminali della grammatica. Questa scelta è estremamente grossolana, dal momento che ci sono alcuni terminali che non si troveranno mai a seguire il lato sinistro delle produzioni per cui si riduce. Più in dettaglio, se riduciamo secondo una produzione $L \to \alpha$ allora è inutile indicare una riduzione per quei terminali che non sono fra i $\mathsf{follow}(L)$ poiché tali terminali non possono mai seguire *L*. Ne consegue che basta mettere le riduzioni *per i soli seguiti di L*. Questa semplice idea dà origine a un parsing più potente di *LR*(0) (nel senso che genera meno conflitti di *LR*(0) e che può quindi essere applicato a più grammatiche).

Definizione 24. Sia *G* una grammatica e sia dato il suo automa *LR*(0). La *tabella SLR* per *G* è identica alla tabella *LR*(0) per *G* (Definizione 22) tranne per il fatto che la seconda regola della Definizione 22 viene sostituita dalla regola:

- per ogni stato *i* che contiene un item $L \to \alpha.$, dove $L \to \alpha$ è la *k*-esima produzione della grammatica, la casella (i, f) della tabella contiene *rk* (*riduci secondo la produzione k*) per tutti gli $f \in \mathsf{follow}(L)$.

Definizione 25. Una grammatica è *SLR* se la sua tabella *SLR* non contiene *conflitti*, cioè caselle con più di un contenuto. Un linguaggio è *SLR* se ha una grammatica *SLR*.

Si noti che il parsing *SLR* ci costringe a calcolare i seguiti (e quindi anche gli annullabili e i primi, con cui i seguiti si calcolano). L'automa a pila è invece lo stesso: cambia solo il modo in cui scriviamo il suo programma (la tabella *SLR*).

Per esempio, la tabella *SLR* per la grammatica in Figura 3.8 è simile alla tabella (3.19) ma contiene meno riduzioni, al punto che non ci sono più conflitti (i seguiti della grammatica in

0) $I \to E\$$
1) $E \to L{=}R$
2) $E \to R$
3) $L \to *R$
4) $L \to \texttt{id}$
5) $R \to L$

Figura 3.12: Una grammatica *LR*(1) ma non *SLR*.

Figura 3.8 sono dati dalla tabella (3.11)):

	$	a	b	*I*	*L*	*A*	*B*
0	*r*2	*s*2	*r*2		*g*1	*g*3	
1	*a*						
2	*r*2	*s*2	*r*2			*g*5	
3	*r*4		*s*6				*g*4
4	*r*1						
5	*r*3		*r*3				
6	*r*4		*s*6				*g*7
7	*r*5						

(3.20)

Si consideri adesso la grammatica in Figura 3.12, che astrae degli assegnamenti in stile C che usano l'operatore * di dereferenziazione. Nella figura le produzioni sono stato numerate in ordine crescente. L'automa *LR*(0) di tale grammatica è mostrato in Figura 3.13. Il calcolo dei nullable, first e follow fornisce le seguenti tabelle:

nullable

	ϕ^0
I	*false*
E	*false*
L	*false*
R	*false*

first

	ϕ^0	ϕ^1	ϕ^2	ϕ^3
I	$\emptyset$	$\emptyset$	$\emptyset$	$\{*, \texttt{id}\}$
E	$\emptyset$	$\emptyset$	$\{*, \texttt{id}\}$	$\{*, \texttt{id}\}$
L	$\emptyset$	$\{*, \texttt{id}\}$	$\{*, \texttt{id}\}$	$\{*, \texttt{id}\}$
R	$\emptyset$	$\emptyset$	$\{*, \texttt{id}\}$	$\{*, \texttt{id}\}$

follow

	ϕ^0	ϕ^1	ϕ^2	ϕ^3
I	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
E	$\emptyset$	{$}	{$}	{$}
L	$\emptyset$	{=}	{=}	{=, $}
R	$\emptyset$	$\emptyset$	{=, $}	{=, $}

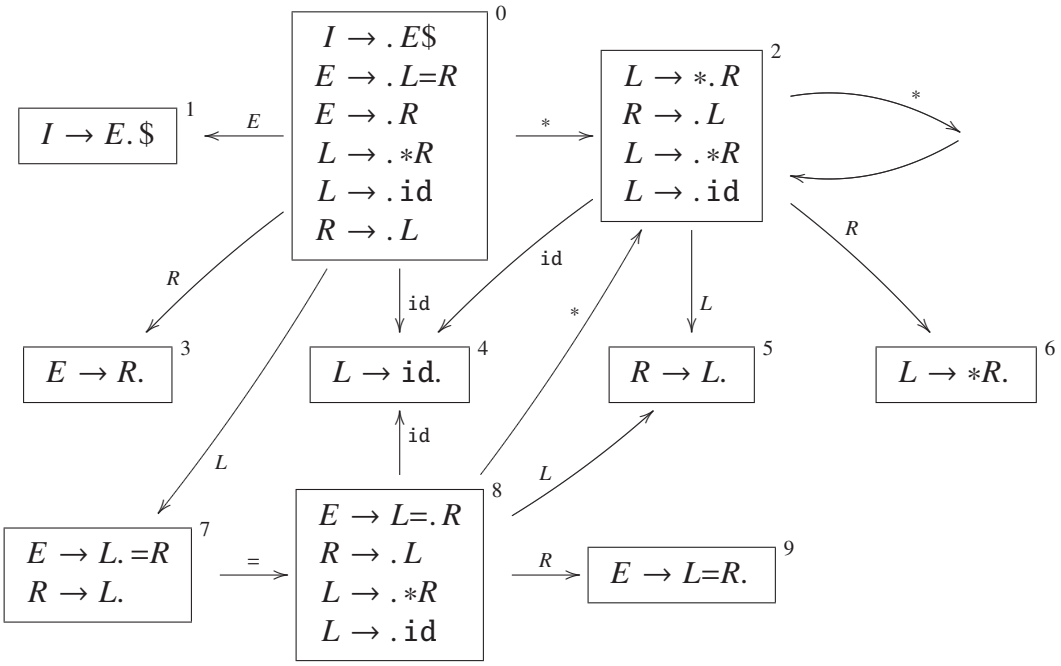


Figura 3.13: L'automa *LR*(0) per la grammatica in Figura 3.12.

Conseguentemente, la tabella *SLR* per la grammatica in Figura 3.12 è

	$	=	∗	id	*I*	*E*	*L*	*R*
0			*s*2	*s*4		*g*1	*g*7	*g*3
1	*a*							
2			*s*2	*s*4			*g*5	*g*6
3	*r*2							
4	*r*4	*r*4						
5	*r*5	*r*5						
6	*r*3	*r*3						
7	*r*5	*s*8/*r*5						
8			*s*2	*s*4			*g*5	*g*9
9	*r*1	*r*1						

Come si vede, c'è un conflitto nello stato 7, per cui la grammatica non è *SLR*. A maggior ragione essa non sarà *LR*(0). Inoltre essa non è neanche *LL*(1) dal momento che gli inizi dei lati destri delle produzioni 1 e 2 sono entrambi {∗, id} e quindi non sono disgiunti.

3.4.3 Il parsing *LR*(1)

Riconsideriamo la grammatica in Figura 3.12, che come abbiamo appena visto non è né *LL*(1), né *LR*(0) né *SLR*. Il motivo per cui non è *SLR* è che nello stato 7 in Figura 3.13 l'item $R \to L.$

richiede una riduzione secondo la produzione $R \to L$ e fra i seguiti di R c'è il carattere =, il che genera un conflitto con la transizione dallo stato 7 allo stato 8 sempre di fronte a tale carattere. Ma lo stato 7 è quello in cui l'automa a pila si trova quando, all'inizio del file sorgente, ha riconosciuto una stringa a cui si può ridurre una L; infatti lo stato 7 è raggiungibile solo dal cammino che parte dallo stato iniziale 0 e va poi in 7 riconoscendo una L. Conseguentemente la R a cui vogliamo ridurre la L tramite l'item $R \to L.$ nello stato 7 è quella che poi nello stato 0 ridurremmo a una E tramite l'item $E \to .\, R$. Ma una E può essere seguita solo da un \$ e mai da un =. In conclusione, sebbene il carattere = sia fra i seguiti di R, esso non può seguire R *nella particolare situazione rappresentata dallo stato* 7.

L'idea del parsing $LR(1)$ che descriviamo in questa sezione è quindi quella di tenere traccia *esplicitamente* di quali seguiti possono realmente seguire i lati sinistri degli item. Inizialmente partiamo dallo stesso item usato nel parsing $LR(0)$, che nel caso della grammatica in Figura 3.12 è $I \to .\, E\$$. Nel chiudere tale item, però, teniamo traccia esplicitamente di quali token ci aspettiamo che possano seguire il non terminale E, indicandoli alla destra di ciascun item aggiunto per formare lo stato. Tali token vengono chiamati *lookahead*. Per esempio, dal momento che nell'item $I \to .\, E\$$ c'è un punto alla immediata sinistra della E, aggiungiamo gli item derivati dalle produzioni per E, usando come possibili lookahead gli inizi di ciò che segue la E che viene dopo il punto dell'item, cioè gli inizi della forma sentenziale \$. Otteniamo quindi i due *item* $LR(1)$

$E \to .\, L{=}R$	\$
$E \to .\, R$	\$

Essi non formano uno stato poiché non sono un insieme chiuso di item. Infatti essi hanno un punto immediatamente alla sinistra di una L e di una R, rispettivamente. Nel primo caso dobbiamo aggiungere gli item derivati dalle produzioni per la L, usando come lookahead gli inizi di ciò che nell'item segue la L, cioè gli inizi di $=R\$$. Nel secondo caso dobbiamo aggiungere gli item derivati dalle produzioni per la R, usando come lookahead gli inizi di ciò che nell'item segue la R, cioè gli inizi di \$. Si noti che in quest'ultimo caso lo stesso lookahead viene usato per capire cosa può seguire la R. Otteniamo l'insieme di item:

$I \to .\, E\$$	
$E \to .\, L{=}R$	\$
$E \to .\, R$	\$
$L \to .\, {*}R$	=
$L \to .\, \texttt{id}$	=
$R \to .\, L$	\$

che *non è ancora uno stato* poiché dobbiamo ancora chiudere rispetto alla L che segue il punto nell'item $R \to .\, L.$ Si noti infatti che, a differenza degli item $LR(0)$ che sono formati da una produzione della grammatica con un punto da qualche parte alla destra, gli item $LR(1)$ hanno anche un lookahead. Conseguentemente, due item che si differenziano solo per il lookahead sono comunque due item diversi. Dobbiamo quindi aggiungere gli item derivati chiudendo $R \to .\, L \quad \$$, cioè quelli derivati dalle produzioni per L usando come lookahead gli inizi di \$. Otteniamo in

conclusione l'insieme di item *LR*(1):

$$
\begin{array}{ll}
I \to .\,E\$ & \\
E \to .\,L{=}R & \$ \\
E \to .\,R & \$ \\
L \to .\,{*}R & = \\
L \to .\,\texttt{id} & = \\
R \to .\,L & \$ \\
L \to .\,{*}R & \$ \\
L \to .\,\texttt{id} & \$
\end{array}
$$

che normalmente viene scritto, in maniera un po' più compatta, come

$$
\begin{array}{ll}
I \to .\,E\$ & \\
E \to .\,L{=}R & \$ \\
E \to .\,R & \$ \\
L \to .\,{*}R & =, \$ \\
L \to .\,\texttt{id} & =, \$ \\
R \to .\,L & \$
\end{array}
$$

Il fatto che gli item *LR*(1) si differenzino anche sulla base del lookahead induce spesso in errori di chiusura, in cui si considerano come stati degli insiemi di item che in effetti non sono chiusi. Occorre prestare particolare attenzione ogni volta che a un insieme di item se ne aggiunge un altro, controllando che questa aggiunta non provochi a sua volta per chiusura l'aggiunta di altri item. Si faccia anche attenzione alla rappresentazione compatta di più item *LR*(1) ottenuta scrivendo insieme i lookahead, come appena visto sopra. Tale scrittura è comunque un'abbreviazione per i due (o più) item distinti per cui, per esempio, quando nella Figura 3.14 dobbiamo chiudere l'item $L \to *.\,R \quad =, \$$ dello stato 2, otteniamo gli item derivati dalle produzioni per R aventi come lookahead gli inizi di quel che segue la R nell'item, cioè *sia* gli inizi di $=$ *che* gli inizi di \$, cioè gli item $R \to .\,L \quad =$ ed $R \to .\,L \quad \$$, che a loro volta scriviamo compattamente come $R \to .\,L \quad =, \$$.

La Figura 3.14 mostra l'*automa LR*(1) per la grammatica in Figura 3.12. Si noti che le transizioni sono ottenute esattamente come nel caso dell'automa *LR*(0), cioè spostando avanti di una posizione i punti degli item e poi chiudendo l'insieme di item così ottenuto. Si noti inoltre che ci sono degli stati uguali *a meno di lookahead*, come gli stati 2 e 13, che vanno comunque considerati distinti poiché item *LR*(1) uguali a meno di lookahead sono item diversi, come già osservato.

Definizione 26. Sia G una grammatica e sia dato il suo automa *LR*(1). La *tabella LR*(1) per G è costruita come la tabella *LR*(0) per G (Definizione 22) a partire però dal suo automa *LR*(1). Inoltre la seconda regola della Definizione 22 viene sostituita dalla regola:

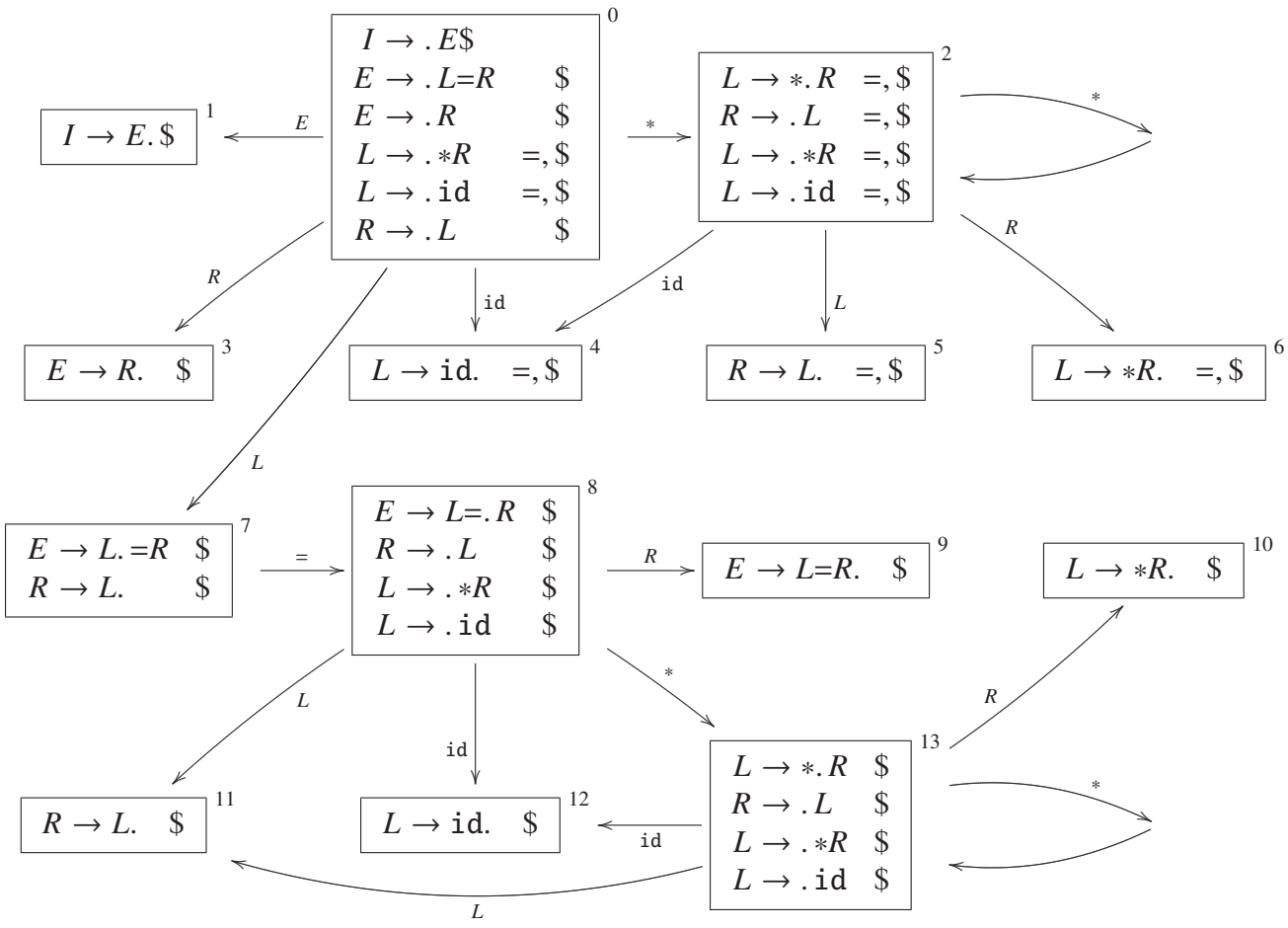


Figura 3.14: L'automa *LR*(1) per la grammatica in Figura 3.12.

- per ogni stato i che contiene un item $L \to \alpha.\ \ f$, dove $L \to \alpha$ è la k-esima produzione della grammatica, la casella (i, f) della tabella contiene rk (*riduci secondo la produzione* k).

Definizione 27. Una grammatica è *LR*(1) se la sua tabella *LR*(1) non contiene *conflitti*, cioè caselle con più di un contenuto. Un linguaggio è *LR*(1) se ha una grammatica *LR*(1).

Si noti che per costruzione i lookahead f in un item $L \to \alpha.\ \ f$ sono fra i seguiti di L. Ne consegue che questo tipo di parsing non genera mai più conflitti del parsing *SLR*. Esso è in effetti strettamente più potente del parsing *SLR* (e quindi per transitività anche del parsing *LR*(0)) poiché la grammatica in Figura 3.12, che come sappiamo non è *SLR*, è invece *LR*(1), come si evince costruendo la sua tabella *LR*(1) come da Definizione 26 e notando che essa non contiene

conflitti:

	$	=	*	id	*I*	*E*	*L*	*R*
0			*s*2	*s*4		*g*1	*g*7	*g*3
1	*a*							
2			*s*2	*s*4			*g*5	*g*6
3	*r*2							
4	*r*4	*r*4						
5	*r*5	*r*5						
6	*r*3	*r*3						
7	*r*5	*s*8						
8			*s*13	*s*12			*g*11	*g*9
9	*r*1							
10	*r*3							
11	*r*5							
12	*r*4							
13			*s*13	*s*12			*g*11	*g*10

Abbiamo quindi ottenuto un metodo di parsing, quello *LR*(1), che sembra sufficientemente potente da essere applicabile a una larga categoria di grammatiche. Purtroppo però il numero di stati dell'automa *LR*(1) è maggiore del numero di stati dell'automa *LR*(0) per la stessa grammatica (la Figura 3.14 contiene 14 stati, contro i 10 stati della Figura 3.13). In effetti, dal momento che stati uguali a meno di lookahead sono comunque da considerarsi distinti, il numero di stati di un automa *LR*(1) può in linea di principio essere esponenziale nel numero di terminali (token) della grammatica. Va detto che questo comportamento è raro, ma sarebbe bello premunirsi di fronte a questa eventualità. Nasce quindi l'idea di trovare un metodo di parsing che generi meno stati del parsing *LR*(1) al prezzo di una piccola riduzione nella potenza di parsing. Questo metodo esiste ed è descritto nella prossima sezione.

3.4.4 Il parsing *LALR*(1) e JavaCup

Consideriamo l'automa in Figura 3.14. Ci sono vari stati uguali a meno di lookahead, come per esempio gli stati 2 e 13. Tali stati portano a loro volta in stati che sono uguali a meno di lookahead. Cosa accade se fondiamo tali stati in un unico stato, unendo l'insieme dei lookahead? Per esempio, fondendo lo stato 2 e lo stato 13 otteniamo lo stato 2 e redirezioniamo ogni freccia entrante in 13 in una freccia entrante in 2. Si può dimostrare che se l'automa risultante non ha conflitti allora esso riconosce esattamente lo stesso linguaggio dell'automa *LR*(1) non semplificato. Inoltre è facile convincersi che questa fusione di stati non può mai introdurre un conflitto sposta/riduci che non c'era già nell'automa *LR*(0), poiché allora ci sarebbe una freccia uscente dallo stato fuso *s* etichettata con un terminale che sta anche fra i lookahead di un item di *s* con il punto alla fine: ma questo implicherebbe che tale conflitto c'era già in almeno uno degli stati fondendo i quali abbiamo ottenuto *s*. È invece possibile introdurre conflitti riduci/riduci, ma essi sono in genere relativamente rari. Inoltre non ricadiamo nel parsing *SLR*, poiché è vero che stiamo fondendo degli stati, ma alcuni stati non potranno essere fusi e manterranno dei lookahead

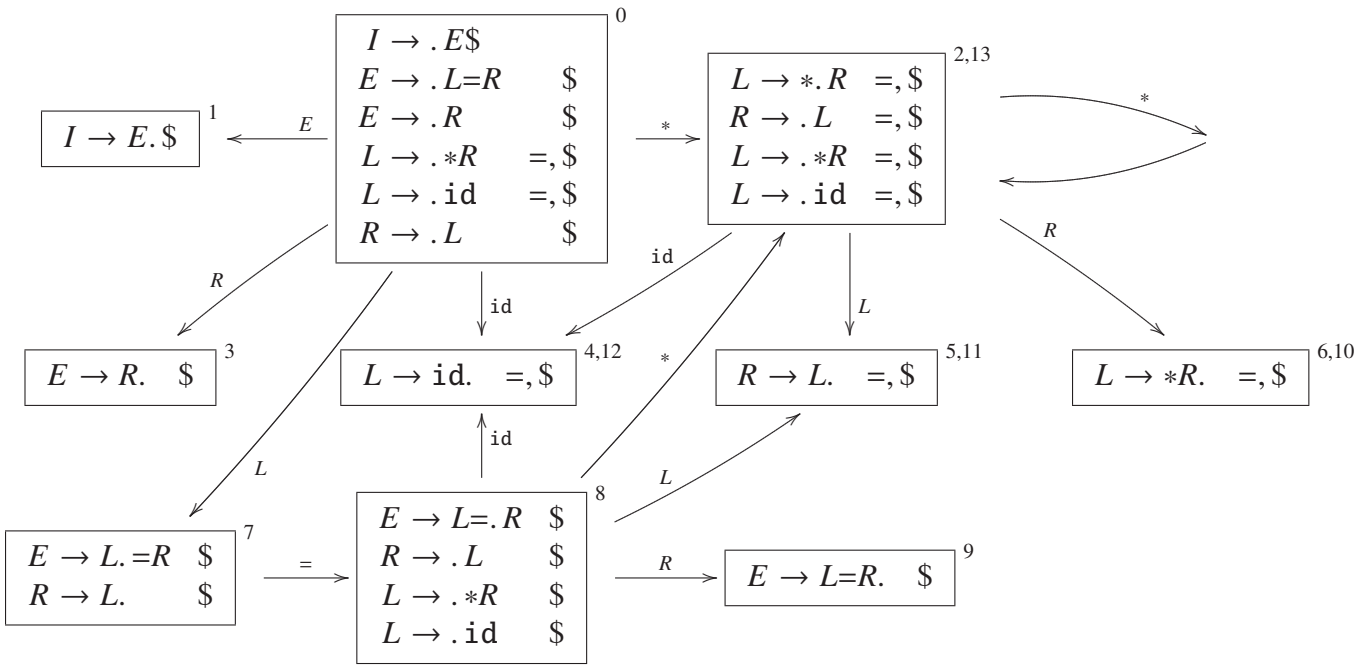


Figura 3.15: L'automa *LALR*(1) per la grammatica in Figura 3.12.

più precisi. E anche gli stati fusi otterranno l'unione dei lookahead, che in genere è un sottoinsieme stretto di *tutti* i seguiti. In conclusione, sembra ragionevole procedere a questa fusione di stati. Nel caso dell'automa in Figura 3.14 otteniamo l'automa semplificato in Figura 3.15, detto *automa LALR*(1) per la grammatica in Figura 3.12.

Definizione 28. Sia G una grammatica e sia dato il suo automa *LALR*(1). La *tabella LALR*(1) per G è costruita come la tabella *LR*(1) per G (Definizione 26) a partire però dal suo automa *LALR*(1).

Definizione 29. Una grammatica è *LALR*(1) se la sua tabella *LALR*(1) non contiene *conflitti*, cioè caselle con più di un contenuto. Un linguaggio è *LALR*(1) se ha una grammatica *LALR*(1).

Per esempio la grammatica in Figura 3.12 è *LALR*(1) poiché la sua tabella *LALR*(1) non contiene

conflitti:

	$	=	*	id	*I*	*E*	*L*	*R*
0			$s(2, 13)$	$s(4, 12)$		$g1$	$g7$	$g3$
1	a							
2, 13			$s(2, 13)$	$s(4, 12)$			$g(5, 11)$	$g(6, 10)$
3	$r2$							
4, 12	$r4$	$r4$						
5, 11	$r5$	$r5$						
6, 10	$r3$	$r3$						
7	$r5$	$s8$						
8			$s(2, 13)$	$s(4, 12)$			$g(5, 11)$	$g9$
9	$r1$							

In questa tabella abbiamo indicato la fusione di due stati con la sequenza degli stati da cui è ottenuta la fusione.

La Figura 3.15 mostra che l'automa *LALR*(1) ha solo 10 stati, in confronto ai 14 dell'automa *LR*(1) in Figura 3.14. Ciò nonostante esso è capace di riconoscere il linguaggio generato dalla grammatica in Figura 3.12. Esistono comunque grammatiche che sono *LR*(1) ma non *LALR*(1), perché la semplificazione dell'automa *LR*(1) introduce dei conflitti riduci/riduci. Conseguentemente il parsing *LALR*(1) è strettamente meno potente del parsing *LR*(1).

Esercizio 17. Si consideri la seguente grammatica:

$$
\begin{aligned}
I &\to A\$ \\
A &\to \mathtt{a}E\mathtt{a} \\
A &\to \mathtt{b}E\mathtt{b} \\
A &\to \mathtt{a}F\mathtt{b} \\
A &\to \mathtt{b}F\mathtt{a} \\
E &\to \\
F &\to
\end{aligned}
$$

Si calcoli il suo automa e tabella *LR*(1) e quindi il suo automa e tabella *LALR*(1). Si concluda che tale grammatica è *LR*(1) ma non *LALR*(1).

Le tecniche di parsing *LR*(0), *LR*(1) ed *LALR*(1) si possono generalizzare a tecniche di parsing che guardano fino a k caratteri davanti alla testina di lettura dell'automa a pila, con $k \geq 0$. Ne segue che esiste una *gerarchia* di tecniche di parsing (e conseguentemente di grammatiche da esse riconosciute). È dimostrabile che ogni grammatica *LL*(k) è anche *LR*(k), per ogni $k \geq 0$, e che il viceversa non è vero. La Figura 3.16 mostra la relazione fra le classi di parsing con $0 \leq k \leq 1$. Si noti che *LALR*(*0*) = *LR*(0). Va osservato inoltre che tutte le classi di grammatica fin qui considerate sono fatte da grammatiche non ambigue. Conseguentemente, nessuna tecnica di parsing fra quelle viste sarà applicabile a una grammatica ambigua. L'ambiguità si traduce infatti in conflitti nella tabella e solo una discesa ricorsiva non deterministica oppure un automa

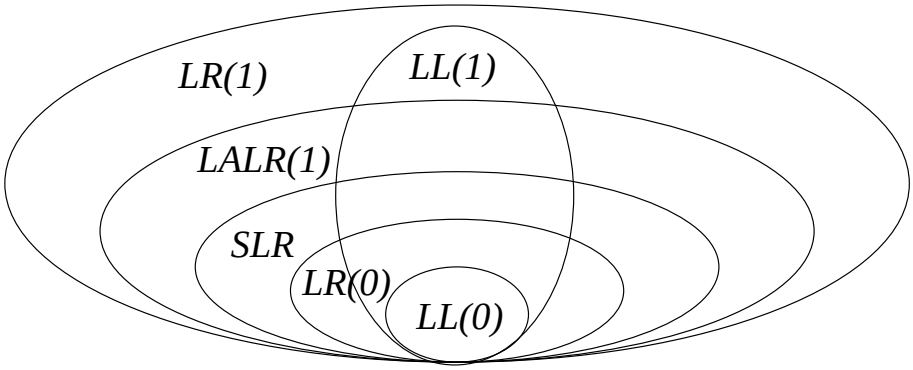


Figura 3.16: La relazione fra le tecniche di parsing e le classi di grammatica considerate.

a pila non deterministico potrebbero seguire al contempo le annotazioni contrastanti della tabella. Ma tali tecniche sarebbero troppo costose in termini computazionali.

Il parsing *LALR*(1) è considerato come il metodo *ideale* di parsing, né troppo costoso né troppo impreciso. Per questo motivo esso è implementato da JavaCup. Va detto che JavaCup costruisce *direttamente* l'automa *LALR*(1), senza passare per la semplificazione dell'automa *LR*(1), evitando quindi l'esplosione combinatoria degli stati per la costruzione dell'automa intermedio *LR*(1). Non ci occupiamo comunque qui di questa ottimizzazione.

È importante invece discutere come si comporta JavaCup se nella costruzione della tabella *LALR*(1) vengono incontrari dei conflitti, situazione non desiderabile ma che purtroppo si verifica spesso in pratica. JavaCup usa in tal caso un sistema di *risoluzione* dei conflitti che consiste nello scegliere una delle annotazioni contrastanti della tabella. Va subito osservato che una simile tecnica in genere restringe l'insieme degli alberi di parsing riconosciuti dall'automa e quindi può potenzialmente cambiare il linguaggio da esso riconosciuto o forzare un'interpretazione piuttosto che un'altra nel caso di grammatiche ambigue. Comunque sia, JavaCup risolve i conflitti sposta/riduci in favore dello spostamento e i conflitti riduci/riduci in favore della riduzione per la produzione che appare prima nella grammatica.

Le scelte di risoluzione dei conflitti incontrati durante la generazione di un parser vengono enumerate da JavaCup nel file di log `syntactical/Kitten.err`, insieme agli stati dell'automa *LALR*(1) e alle relative transizioni. Tale file andrebbe quindi sempre controllato dopo la generazione di un parser. È possibile specificare un numero massimo di risoluzioni accettabili da JavaCup, superato il quale la creazione del parser non è effettuata.

3.4.5 Il parsing *LR* con grammatiche ambigue

Abbiamo osservato che nessuna grammatica ambigua può essere processata con uno dei metodi di parsing già visti. Abbiamo anche detto che è spesso possibile trovare grammatiche non ambigue equivalenti, ma che esse sono tipicamente complesse e innaturali (Sezione 3.2.4). In questa sezione riconsideriamo il problema partendo dalla grammatica in Figura 3.17 che esprime in piccolo i problemi di ambiguità della grammatica per le espressioni Kitten vista nella Sezione 3.2.3.

0) $I \rightarrow exp\ \$$
1) $exp \rightarrow exp$ `PLUS` exp
2) $exp \rightarrow exp$ `TIMES` exp
3) $exp \rightarrow$ `INTEGER`

Figura 3.17: Una grammatica ambigua per le espressioni aritmetiche.

La Figura 3.18 mostra l'automa *LR*(1) per la grammatica in Figura 3.17. Conseguentemente la sua tabella *LR*(1) è la seguente, in cui sono evidenti molti conflitti:

	\$	PLUS	TIMES	INTEGER	*I*	*exp*
0				*s*4		*g*6
1	*r*2	*s*3/*r*2	*s*5/*r*2			
2	*r*1	*s*3/*r*1	*s*5/*r*1			
3				*s*4		*g*2
4	*r*3	*r*3	*r*3			
5				*s*4		*g*1
6	*a*	*s*3	*s*5			

(3.21)

C'è un conflitto sposta/riduci nello stato 1, di fronte al token `PLUS`, poiché possiamo sia spostarci nello stato 3 che ridurre secondo la produzione $exp \rightarrow exp$ `TIMES` exp. L'item $exp \rightarrow exp$ `TIMES` exp. nello stato 1 ci dice che in tale stato abbiamo finito di leggere dal file sorgente qualcosa che è il prodotto di due espressioni exp_1 ed exp_2. Ridurre secondo la produzione $exp \rightarrow exp$ `TIMES` exp significherebbe quindi vedere tale prodotto come un'unica espressione il cui risultato è sommato con quel che segue. Spostare il token `PLUS` significherebbe considerare exp_2 come l'inizio di una addizione, il cui risultato deve essere poi moltiplicato per exp_1. È qui evidente che ci scontriamo contro l'ambiguità della grammatica. Ridurre secondo la produzione $exp \rightarrow exp$ `TIMES` exp significa dare priorità alla moltiplicazione, mentre spostare `PLUS` significa dare priorità all'addizione. La scelta ragionevole è quindi quella di risolvere l'ambiguità riducendo secondo la produzione $exp \rightarrow exp$ `TIMES` exp. In termini della tabella *LR*(1), questo significa che nello sttao 1, di fronte a `PLUS`, risolviamo il conflitto lasciando l'azione di riduzione ed eliminado l'azione di spostamento del token. Un ragionamento simile ci fa concludere che nello stato 2, di fronte al token `TIMES`, preferiamo spostare il token piuttosto che ridurre secondo la produzione $exp \rightarrow exp$ `PLUS` exp.

Un altro conflitto sorge ancora nello stato 1 di fronte al token `TIMES`. In tale stato abbiamo già letto dei token che formano la moltiplicazione di due espressioni exp_1 ed exp_2. Abbiamo sia la possibilità di ridurre secondo la produzione $exp \rightarrow exp$ `TIMES` exp che di spostare il token `TIMES` e andare nello stato 5. La prima scelta significa legare il prodotto di exp_1 ed exp_2 riducendolo a un'espressione moltiplicata per quel che segue il `TIMES`, mentre la seconda scelta considera exp_2 come l'inizio di un prodotto il cui risultato viene moltiplicato per exp_1. Dal momento che preferiamo una associatività a sinistra per la moltiplicazione, facciamo la scelta di ridurre

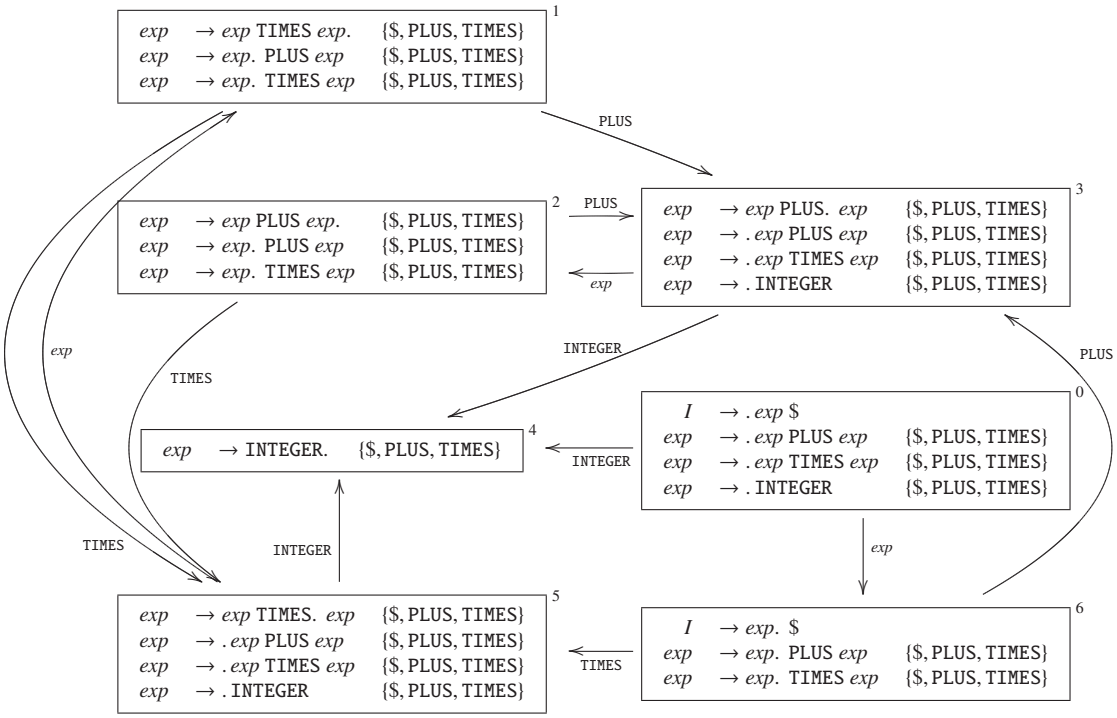


Figura 3.18: L'automa *LR*(1) per la grammatica in Figura 3.17.

secondo la produzione *exp* → *exp* `TIMES` *exp*. Similmente nello stato 2 di fronte al token `PLUS` preferiamo ridurre secondo la produzione *exp* → *exp* `PLUS` *exp* piuttosto che spostare e andare nello stato 3. Ecco quindi che la tabella (3.21) viene semplificata in una tabella senza conflitti:

	$	PLUS	TIMES	INTEGER	*I*	*exp*
0				*s*4		*g*6
1	*r*2	*r*2	*r*2			
2	*r*1	*r*1	*s*5			
3				*s*4		*g*2
4	*r*3	*r*3	*r*3			
5				*s*4		*g*1
6	*a*	*s*3	*s*5			

che implementa il parsing delle espressioni aritmetiche con le usuali regole di precedenza e associatività.

La specifica della precedenza e dell'associatività degli operatori aritmetici viene fatta in Java-Cup con le direttive che abbiamo visto nella Sezione 3.2.4, le quali modificano il comportamento di JavaCup nella risoluzione dei conflitti (Sezione 3.4.4). Una direttiva `precedence xxx` *t* dà infatti al token *t* una priorità maggiore di quella di tutti gli altri token enumerati dalle direttive precedenti. Inoltre essa dà alle produzioni il cui ultimo token a destra è *t* una priorità pari a quella di *t*. Un conflitto sposta/riduci viene a questo punto risolto preferendo lo spostamento se

il token spostato ha priorità maggiore della produzione per cui si dovrebbe ridurre; la riduzione nel caso opposto. Conseguentemente, con le direttive della Sezione 3.2.4 fra una riduzione per *exp* → *exp* `TIMES` *exp* e lo spostamento di un `PLUS` si preferisce la riduzione. A parità di priorità si seguono le direttive di associatività preferendo la riduzione se l'associatività è `left`, lo spostamento se l'associatività è `right` e lasciando la casella vuota se l'associatività è `nonassoc`, in modo da segnalare un errore in tale situazione.

Un altro problema di ambiguità della grammatica Kitten (e in genere di tutti i linguaggi imperativi) è relativo all'`if/then/else`, in cui il ramo `else` è normalmente facoltativo. Ne consegue che nel caso di `if` annidati risulta ambigua l'associazione degli `else` all'`if` da cui dipendono. Questo problema è tipicamente risolto associando ogni `else` all'ultimo `then` incontrato. Per esempio, vogliamo che `if (a > 5) then if (b < 4) then a := 3 else b := 6` venga interpretato come `if (a > 5) then {if (b < 4) then a := 3 else b := 6}` piuttosto che come `if (a > 5) then {if (b < 4) then a := 3} else b := 6`. A tal fine il parser, di fronte all'ultimo token `ELSE`, deve spostare tale token piuttosto che ridurre secondo la produzione

```
exp ::= IF LPAREN exp RPAREN THEN command
```

della Sezione 3.2.5. Abbiamo detto nella Sezione 3.4.4 che JavaCup risolve un conflitto sposta/riduci in favore dello spostamento, che è quello che volevamo, e annotando nel file `syntactical/Kitten.` che il conflitto è stato risolto in tal senso. Per evitare tale annotazione (essenzialmente un *warning*) e non contare questa risoluzione nel novero di quelle ammesse al massimo da JavaCup, basta dichiarare esplicitamente che l'`ELSE` ha priorità rispetto al `THEN`. Otteniamo questo effetto aggiungendo al file `syntactical/Kitten.cup` le dichiarazioni:

```
precedence nonassoc THEN;
precedence nonassoc ELSE;
```

la cui annotazione di associatività è irrilevante. Un simile problema si presenta fra gli operatori di confronto e i token `DOT` e `LBRACK`, risolto in modo simile (si veda la Sezione 3.2.4).

Un altro problema di ambiguità della grammatica Kitten della Sezione 3.2 è legato al meno unario. L'espressione `MINUS` *exp* `PLUS` *exp* può essere interpretata sia come `MINUS` (*exp* `PLUS` *exp*) che come (`MINUS` *exp*) `PLUS` *exp* e quest'ultima è l'interpretazione preferita. Conseguentemente la riduzione secondo la produzione `exp ::= MINUS exp` deve essere preferita a qualsiasi spostamento dei token che seguono la prima espressione. Otteniamo questo effetto dando esplicitamente una priorità massima a tale produzione:

```
exp ::= MINUS exp %prec UMINUS
```

dove il token `UMINUS` ha ricevuto una priorità maggiore di qualsiasi suo seguito (Sezione 3.2.4).

Risolti questi aspetti di ambiguità della grammatica Kitten, il programma JavaCup è capace di generare il parser per Kitten senza segnalare alcuna risoluzione di conflitto.

Concludiamo questa sezione ricordando che la risoluzione dei conflitti tramite annotazioni di precedenza e associatività è generalmente pericolosa perché si rischia di cambiare il linguaggio riconosciuto dal parser. Essa è usata in letteratura limitatamente ai soli esempi visti in questa sezione.

3.5 Le azioni semantiche e la costruzione induttiva della sintassi astratta

La grammatica Kitten della Sezione 3.2 specifica quali stringhe (*file sorgenti*) appartengono al linguaggio Kitten. Il parser generato da JavaCup si limita quindi a riconoscere le stringhe del linguaggio. JavaCup ammette però la possibilità di *decorare* la grammatica con delle *azioni semantiche* che vengono eseguite in corrispondenza alle azioni di riduzione della tabella *LALR*(1). Tali azioni semantiche possono essere usate per molti scopi. In questa sezione vediamo alcuni esempi.

Riconsideriamo la grammatica in Figura 3.8, che in JavaCup è scritta come

```
terminal a b;
non terminal L A B;

start with L;

L ::= A B ;
A ::=
 | a A ;
B ::=
 | b B ;
```

Supponiamo di voler sapere, per ogni file sorgente, non solo se esso soddisfa la grammatica, cioè se esso è formato da una lista di `a` seguita da una lista di `b`, ma anche la lunghezza delle due liste. A tal fine decidiamo che il non terminale *A* deve conoscere quante `a` sono state derivate da esso e il non terminale *B* quante `b` sono state derivate da esso. Diciamo che il *valore semantico* del non terminale *A* è il numero di `a` da esso derivate e il valore semantico del non terminale *B* è il numero di `b` da esso derivate. I valori semantici vanno dichiarati nella enumerazione dei non terminali. Dal momento che nel nostro caso si tratta di valori interi, scriveremo[4]

```
non terminal int A;
non terminal int B;
```

A questo punto dobbiamo specificare come si calcolano tali valori semantici. Il calcolo avviene *decorando* ciascuna produzione per *A* con delle *azioni semantiche* che specificano il valore semantico di *A* per ciascuna delle sue due produzioni. Similmente per *B*:

```
A ::=
   {: RESULT = 0; :}
 | a A:l
   {: RESULT = 1 + l; :} ;
```

[4]Il valore semantico in JavaCup deve in effetti essere un oggetto, per cui non è possibile utilizzare il tipo primitivo `int` ma occorrerebbe far ricorso alla classe involucro `java.lang.Integer`. È solo per semplicità espositiva che preferiamo utilizzare nei nostri esempi il tipo `int`.

```
B ::=
   {: RESULT = 0; :}
 | b B:l
   {: RESULT = 1 + l; :} ;
```

Le azioni semantiche sono codice Java che si aggiunge dopo ciascuna produzione, racchiuso fra i delimitatori `{:` e `:}`. Tale codice calcola il valore semantico `RESULT` usando i valori semantici dei componenti dei lati destri delle produzioni. Nell'esempio sopra diciamo che se una lista di `a` è vuota allora il numero di `a` incontrate è 0. Se una lista di `a` è invece fatta da una `a` seguita da l `a`, il numero complessivo di `a` incontrate è $1 + l$. Un ragionamento simile si applica per B. Si noti che abbiamo *decorato* dei non terminali alla destra delle produzioni facendoli seguire da un carattere due punti e da una variabile che contiene il loro valore semantico. È possibile decorare anche i terminali che stanno alla destra di una produzione. Il valore semantico dei terminali è per definizione il loro valore lessicale (Capitolo 2) che normalmente è `null` tranne se l'analizzatore lessicale ha sintettizato per essi un apposito valore lessicale, come avviene in Kitten per gli identificatori, le stringhe e le costanti numeriche.

Continuando il nostro esempio, il numero di `a` e il numero di `b` incontrate nel file sorgente vanno fatti risalire fino al non terminale iniziale. Dal momento che si tratta di *due* interi, siamo costretti a definire una struttura dati composta da due campi di tipo `int`:

```
public class Pair {
  private int a;
  private int b;

  public Pair(int a, int b) {
    this.a = a;
    this.b = b;
  }
}
```

Dichiariamo il tipo del valore lessicale per la L:

```
non terminal Pair L;
```

quindi specifichiamo come si costruisce tale valore lessicale:

```
L ::= A:a B:b
    {: RESULT = new Pair(a,b); :} ;
```

La grammatica decorata è in Figura 3.19. Il valore semantico del non terminale iniziale è poi ritornato come valore di ritorno del metodo `parse()` della classe `Parser` che viene generata da JavaCup (Sezione 3.2.7).

L'implementaziome delle azioni semantiche è basata su una semplice modifica dell'automa a pila della Sezione 3.4. Oltre a utilizzare uno stack di stati, l'automa a pila utilizza adesso

```
terminal a b;
non terminal int A;
non terminal int B;
non terminal Pair L;
start with L;

L ::= A:a B:b
     {: RESULT = new Pair(a,b); :} ;

A ::=
     {: RESULT = 0; :}
  | a A:l
     {: RESULT = 1 + l; :} ;

B ::=
     {: RESULT = 0; :}
  | b B:l
     {: RESULT = 1 + l; :} ;
```

Figura 3.19: La grammatica di Figura 3.8 decorata con delle azioni semantiche che calcolano il numero di a e il numero di b incontrate nel file sorgente.

anche uno stack di valori semantici, corrispondenti ai terminali o non terminali che sono stati spostati o a cui si è ridotto per ottenere lo stato nella posizione corrispondente dello stack di stati. Tale stack di valori semantici è in effetti implementato da JavaCup come uno stack di `java_cup.runtime.Symbol` (Figura 2.2). Il campo `value` è utilizzato proprio per contenere il valore semantico ed è accessibile tramite la variabile *v* che si dichiara nella notazione *terminale* : *v* o *non terminale* : *v*.

Simuliamo per esempio il comportamento dell'automa a pila di fronte alla stringa aab$, utilizzando la tabella (3.20) e le azioni semantiche in Figura 3.19. Indicando con / il valore semantico `null`, la configurazione iniziale dell'automa è:

0	aab$
/	

dove il valore semantico / per lo stato 0 è irrilevante. A questo punto, di fronte al lookahead a, la tabella (3.20) ci dice di andare nello stato 2. Dal momento che il valore semantico dei token è per default `null`, otteniamo la configurazione

0, 2	ab$
/, /	

Nello stato 2 di fronte al lookahead a restiamo in 2:

$$\begin{array}{ll} 0,2,2 & \texttt{b\$} \\ /,/,/ & \end{array}$$

mentre di fronte al lookahead b riduciamo secondo la produzione $A \to \varepsilon$ e poi andiamo nello stato 5. La produzione è stata decorata in modo tale che il valore semantico della A è 0. La configurazione risultante è quindi:

$$\begin{array}{ll} 0,2,2,5 & \texttt{b\$} \\ /,/,/,0 & \end{array}$$

Nello stato 5 di fronte al lookahead b riduciamo secondo la produzione $A \to \texttt{a}A$ per cui dobbiamo levare due stati dallo stack e sostituirli con lo stato 5. Gli ultimi due elementi dello stack dei valori semantici sono / e 0 per cui nella Figura 3.19 il valore di l è 0. Conseguentemente il valore semantico $1 + l$ è pari ad 1 e otteniamo la configurazione:

$$\begin{array}{ll} 0,2,5 & \texttt{b\$} \\ /,/,1 & \end{array}$$

Dobbiamo nuovamente ridurre secondo la produzione $A \to \texttt{a}A$ ottenendo questa volta:

$$\begin{array}{ll} 0,3 & \texttt{b\$} \\ /,2 & \end{array}$$

Nello stato 3 di fronte al lookahead b finiamo nello stato 6:

$$\begin{array}{ll} 0,3,6 & \$ \\ /,2,/ & \end{array}$$

e nello stato 6 di fronte al lookahead \$ riduciamo secondo la produzione $B \to \varepsilon$ per cui otteniamo la configurazione

$$\begin{array}{ll} 0,3,6,7 & \$ \\ /,2,/,0 & \end{array}$$

Nello stato 7 di fronte al lookahead \$ riduciamo secondo la produzione $B \to \texttt{b}B$ per cui dobbiamo eliminare due stati dallo stack e sostituirli con lo stato 4. Inoltre avremo $l = 0$ in Figura 3.19 e conseguentemente otteniamo la configurazione:

$$\begin{array}{ll} 0,3,4 & \$ \\ /,2,1 & \end{array}$$

```
terminal PLUS, TIMES;
terminal int INTEGER;
non terminal int exp;
start with exp;

exp ::=
    exp:e1 PLUS exp:e2
      {: RESULT = e1 + e2; :}
  | exp:e1 TIMES exp:e2
      {: RESULT = e1 * e2; :}
  | INTEGER:i
      {: RESULT = i; :} ;
```

Figura 3.20: La grammatica di Figura 3.17 decorata con delle azioni semantiche che calcolano il valore dell'espressione di cui il file sorgente è composto.

Nello stato 4 di fronte al lookahead \$ dobbiamo ridurre secondo la produzione $L \rightarrow AB$ per cui dobbiamo eliminare due stati dallo stack e sostituirli con lo stato 1. In Figura 3.19 avremo $a = 2$ e $b = 1$ per cui otteniamo la configurazione

$$\begin{array}{ll} 0, 1 & \$ \\ /, p & \end{array}$$

dove p è un puntatore in memoria a un oggetto `Pair` i cui campi `a` e `b` contengono rispettivamente 2 e 1. A questo punto l'automa si ferma accettando la stringa `aab`\$ poiché nello stato 1 di fronte al lookahead \$ la tabella 3.20 richiede di accettare il file sorgente.

Consideriamo un altro esempio di decorazione di una grammatica con azioni semantiche. La grammatica in Figura 3.17 specifica delle espressioni aritmetiche su interi. Supponiamo che l'analizzatore lessicale associ al token `INTEGER` il valore numerico concreto presente nel file sorgente (Capitolo 2). Le azioni semantiche in Figura 3.20 calcolano il valore dell'espressione contenuta nel file sorgente. Si noti che se l'espressione è formata semplicemente da un numero intero allora la produzione decorata

```
exp ::= INTEGER:i
     {: RESULT = i; :} ;
```

usa il valore lessicale del token `INTEGER` per sintetizzare il valore semantico di `exp`. In tal caso occorre dichiarare qual è il valore lessicale di `INTEGER`, con la dichiarazione

```
terminal int INTEGER;
```

Tale dichiarazione deve essere compatibile con il tipo del valore lessicale effettivamente calcolato dall'analizzatore lessicale per il token `INTEGER`.

```
terminal a b;
non terminal AbstractA A;
non terminal AbstractB B;
non terminal AB L;
start with L;

L ::= A:a B:b
     {: RESULT = new AB(a,b); :} ;

A ::=
     {: RESULT = new EmptyA(); :}
  | a A:l
     {: RESULT = new OneA(l); :} ;

B ::=
     {: RESULT = new EmptyB(); :}
  | b B:l
     {: RESULT = new OneB(l); :} ;
```

Figura 3.21: La grammatica di Figura 3.8 decorata con delle azioni semantiche che sintetizzano la sua sintassi astratta.

Esercizio 18. Si scriva una grammatica non ambigua che genera il linguaggio delle stringhe di `a` e `b`. Quindi la si decori con delle azioni semantiche che calcolano la differenza fra il numero delle `a` e il numero delle `b`.

Esercizio 19. Supponendo che il token `INTEGER` rappresenti solo numeri interi maggiori o uguali a 0, si decori la grammatica della Figura 3.17 con delle azioni semantiche che calcolano un valore booleano. Tale valore deve essere *true* se e solo se il valore dell'espressione non è 0.

Un'applicazione delle azioni semantiche è la creazione, durante il parsing, della *sintassi astratta* del codice sorgente, cioè di un albero, come quello della Figura 3.1, che descrive la *struttura logica* del codice. L'idea è quella di fare sintetizzare a ciascun non terminale, come valore semantico, la sintassi astratta della parte di codice da esso derivata.

Supponiamo per esempio di volere generare la sintassi astratta per la grammatica in Figura 3.8, modificando le azioni semantiche della Figura 3.19. Otteniamo la grammatica decorata in Figura 3.21. La classe `EmptyB` rappresenta una sequenza vuota di `b`. La classe `OneB` rappresenta invece una `b` seguita da una sequenza di `b`. Dal momento che dobbiamo assegnare *un* tipo al valore semantico sintetizzato per `B`, tali due classi devono essere sottoclassi di una classe `AbstractB` che denota genericamente delle sequenze di `b`. Tale classe è bene che sia lasciata astratta, nel senso di Java:

```
public abstract class AbstractB {}
```

```
public class EmptyB extends AbstractB {}

public class OneB extends AbstractB {
  private AbstractB l;
  public OneB(AbstractB l) { this.l = l; }
}
```

Identica è l'impostazione delle classi `EmptyA`, `OneA` e `AbstractA`. La classe `AB` è invece definita come:

```
public class AB {
  private AbstractA a;
  private AbstractB b;
  public AB(AbstractA a, AbstractB b) { this.a = a; this.b = b; }
}
```

dal momento che c'è solo una produzione per `L`.

Si noti l'estrema *arbitrarietà* della rappresentazione della sintassi astratta. Per esempio, un'altra possibile organizzazione della sintassi astratta per la grammatica in Figura 3.8 è mostrata in Figura 3.22. Questa volta le classi di sintassi astratta sono implementate come

```
public class ListA {
  private ListA tail;
  public ListA(ListA tail) { this.tail = tail; }
}

public class ListB {
  private ListB tail;
  public ListB(ListB tail) { this.tail = tail; }
}

public class AB {
  private ListA a;
  private ListB b;
  public AB(ListA a, ListB b) { this.a = a; this.b = b; }
}
```

Altre scelte sarebbero possibili e legittime. In genere è importante che la sintassi astratta semplifichi la comprensione e l'elaborazione del codice che essa astrae (Capitolo 4). Una buona euristica è quella di definire una classe di sintassi astratta per ogni produzione, i cui oggetti hanno un campo per ogni non terminale nel lato destro della produzione. Quindi si definisce una classe astratta (nel senso di Java) che fa da superclasse a tutte le classi di sintassi astratta per le produzioni che hanno a sinistra lo stesso non terminale. Da questo punto di vista è quindi più *standard* una sintassi astratta generata come in Figura 3.21 che non una generata come in Figura 3.22.

```
terminal a b;
non terminal ListA A;
non terminal ListB B;
non terminal AB L;
start with L;

L ::= A:a B:b
     {: RESULT = new AB(a,b); :} ;

A ::=
     {: RESULT = null; :}
  | a A:l
     {: RESULT = new ListA(l); :} ;

B ::=
     {: RESULT = null; :}
  | b B:l
     {: RESULT = new ListB(l); :} ;
```

Figura 3.22: La grammatica di Figura 3.8 decorata con delle azioni semantiche che sintetizzano la sua sintassi astratta come liste di a e di b.

Le azioni semantiche possono essere utilizzate per svariati scopi. L'unico uso per cui le utilizziamo nel compilatore Kitten è per la generazione della sintassi astratta del codice sorgente. Su tale sintassi astratta definiamo poi dei metodi virtuali a discesa ricorsiva che permettono per esempio di effettuare il type-checking e la generazione del codice intermedio (Capitoli 5 e 6). È possibile comunque utilizzare le stesse azioni semantiche per svolgere tali compiti. Questo approccio è sicuramente più tradizionale [1] ma finisce per sovraccaricare il file `syntactical/Kitten.cup` con informazione non relativa all'aspetto sintattico del linguaggio. Inoltre l'uso di un linguaggio a oggetti per l'implementazione del compilatore ben si accompagna alla definizione del type-checking e della generazione del codice intermedio tramite metodi virtuali delle classi di sintassi astratta, permettendo per esempio di definire in maniera molto semplice un comportamento di default per tutta una classe di strutture sintattiche (come per gli operatori binari).

3.6 La sintassi astratta di Kitten

La generazione della sintassi astratta di Kitten avviene come abbiamo visto sopra in Figura 3.21. L'idea è di far sintetizzare a ciascun non terminale, tramite azioni semantiche, l'albero di sintassi astratta della parte di codice sorgente da esso derivato.

Vediamo per esempio come modifichiamo a tal fine una delle produzioni della Sezione 3.2.3:

```
exp ::= exp:left PLUS:p exp:right
   {: RESULT = new Addition(pleft,left,right); :}
```

Per induzione, `left` e `right` contengono l'albero di sintassi astratta per la parte di codice derivata dai due addendi dell'addizione. Invece `p` contiene il valore lessicale del token `PLUS`, che come abbiamo già detto è `null` essendo `PLUS` un terminale. Questo non significa che la notazione `PLUS:p` sia inutile: essa dichiara implicitamente anche una variabile `pleft` che dice quanti caratteri sono passati dall'inizio del file sorgente fino al token `PLUS`. In pratica, `pleft` è un accesso al campo `left` della struttura dati in Figura 2.2. Conservare questa informazione nell'albero astratto è importante nel caso in cui, in futuro, servisse segnalare un qualche errore su questa addizione (Capitolo 2). Si noti che esistono anche le variabili `leftleft` corrispondente a `left` e `rightleft` corrispondente a `right`, ma in questo caso esse non sono utilizzate. Quello che stiamo dicendo con la precedente produzione è quindi che il valore semantico per una addizione è un albero astratto con una radice che è un nodo di tipo `Addition` e i cui due figli sono gli alberi astratti per i due addendi dell'addizione. Inoltre la posizione in cui deve essere segnalato un eventuale errore semantico è quella del token `PLUS`.

Affiché la definizione induttiva dell'albero astratto per un pezzo di codice sia ben fondata, occorre che ci siano anche dei casi base. Per esempio, un caso base è il seguente:

```
exp ::= TRUE:t
   {: RESULT = new True(tleft); :}
```

il quale dice che il valore semantico per la costante `true` è un nodo di tipo `True`, privo di figli. Eventuali errori su questa parte di codice devono essere in futuro segnalati alla posizione in cui inizia l'espressione `true`, cioè a `tleft` caratteri dall'inizio del file sorgente.

Le classi di sintassi astratta utilizzate per rappresentare il codice sorgente Kitten in maniera strutturata si trovano all'interno della directory `absyn` di Kitten. Esse sono tutte sottoclassi della classe astratta (nel senso di Java) `absyn/Absyn.java` mostrata in Figura 3.23. Una classe di sintassi astratta ha sempre un campo `pos` che indica dove deve essere segnalato un errore verificatosi sulla parte di codice da essa rappresentata. La posizione `pos` viene specificata al momento della creazione del nodo di sintassi astratta tramite le azioni semantiche e può essere letta in seguito con il metodo `getPos()`. Si noti che ogni nodo di sintassi astratta ha anche un identificatore numerico unico `identifier`, la cui utilità sarà chiara in seguito quando descriveremo la rappresentazione grafica dell'albero di sintassi astratta (Sezione 4.4).

Le espressioni sono una sottoclasse di `absyn/Absyn.java`. Le definiamo come

```
public abstract class Expression extends Absyn {
  protected Expression(int pos) {
    super(pos);
  }
}
```

A questo punto possiamo dire che la classe di sintassi astratta `absyn/True.java` è un caso particolare di espressione:

```
public abstract class Absyn {
  private int pos;
  private int identifier;
  private static int counter = 0;

  protected Absyn(int pos) {
    this.pos = pos;
    this.identifier = counter++;
  }

  public int getPos() {
    return pos;
  }
}
```

Figura 3.23: La superclasse di tutte le classi di sintassi astratta per Kitten.

```
public class True extends Expression {
  public True(int pos) {
    super(pos);
  }
}
```

Si noti che questa volta si tratta di una classe concreta, nel senso di Java.

Il caso della classe di sintassi astratta `absyn/Addition.java`, che rappresenta un'operazione binaria di addizione, è più complesso. In primo luogo, definiamo le operazioni binarie come un caso particolare delle espressioni:

```
public abstract class BinOp extends Expression {
  private Expression left;
  private Expression right;

  protected BinOp(int pos, Expression left, Expression right) {
    super(pos);
    this.left = left;
    this.right = right;
  }
}
```

Si noti che un'operazione binaria ha due campi `left` e `right` che sono, ricorsivamente, la sintassi astratta dei suoi due operandi. Si noti inoltre che il costruttore inizializza la parte di stato di sua competenza e demanda alla superclasse l'inizializzazione del resto, cioè in questo caso di `pos`. A questo punto definiamo un caso particolare di operazione binaria, cioè un'operazione binaria aritmetica:

```
public abstract class ArithmeticBinOp extends BinOp {
  protected ArithmeticBinOp(int pos, Expression left, Expression right) {
    super(pos,left,right);
  }
}
```

Siamo finalmente nelle condizioni di definire `absyn/Addition.java` come un caso particolare di operazione binaria aritmetica:

```
public class Addition extends ArithmeticBinOp {
  public Addition(int pos, Expression left, Expression right) {
    super(pos,left,right);
  }
}
```

Questa volta si tratta di una classe concreta, nel senso di Java. Si osservi che le classi astratte, nel senso di Java, hanno costruttori `protected`, utilizzabili quindi solo dalle classi concrete che le estendono, tramite la chiamata `super` a un costruttore della superclasse.

La strutturazione gerarchica delle classi di sintassi astratta e l'uso intenso di classi astratte, nel senso di Java, può non essere immediatamente apprezzabile. Quando, però, definiremo algoritmi ricorsivi sulla sintassi astratta, ci accorgeremo che una buona strutturazione gerachica aiuta significativamente la definizione di tali algoritmi. È un tipico caso in cui l'impostazione a oggetti del codice semplifica nettamente lo sviluppo del software.

Vediamo adesso in maniera più dettagliata quali sono le classi di sintassi astratta di Kitten.

3.6.1 Le classi di sintassi astratta per i tipi

La Figura 3.24 mostra la gerarchia delle classi di sintassi astratta per le espressioni di tipo dei programmi Kitten. Indichiamo con un ovale una classe astratta (nel senso di Java) e con un rettangolo una classe concreta. Queste classi vengono istanziate dalle produzioni che definiscono i tipi Kitten (si confronti con la Sezione 3.2.2):

```
type ::=
   ID:id
   {: RESULT = new ClassTypeExpression(idleft,Symbol.mk(id)); :}
 | BOOLEAN:b
   {: RESULT = new BooleanTypeExpression(bleft); :}
 | INT:i
   {: RESULT = new IntTypeExpression(ileft); :}
 | FLOAT:f
   {: RESULT = new FloatTypeExpression(fleft); :}
 | ARRAY:a OF type:t
```

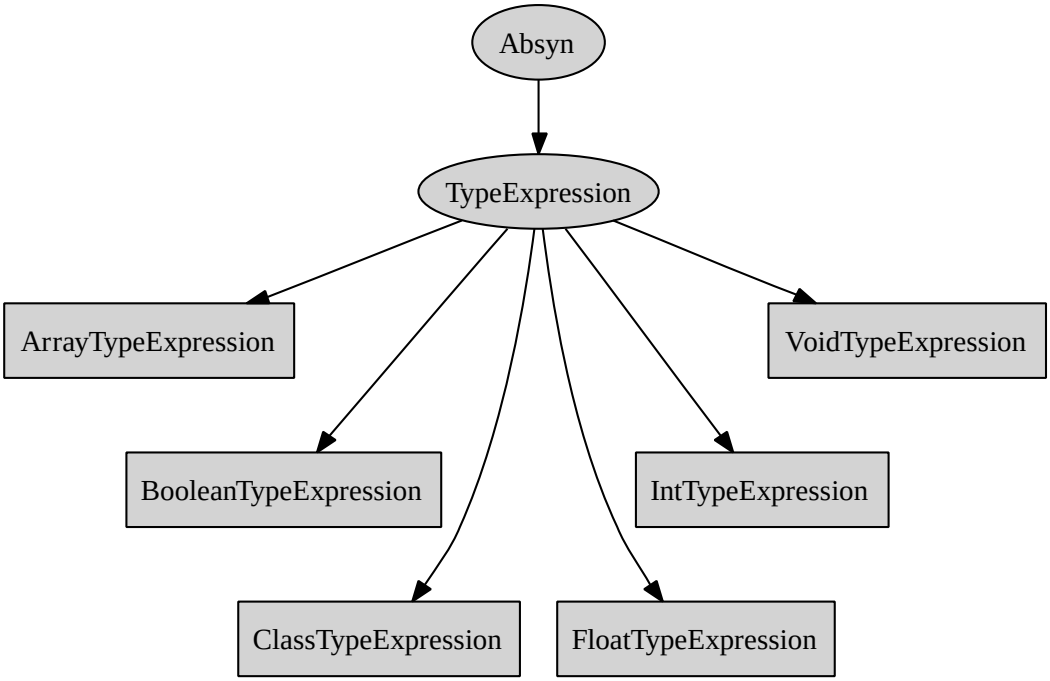


Figura 3.24: La struttura gerarchica delle classi di sintassi astratta per i tipi. Le classi astratte sono indicate con un ovale, quelle concrete con un rettangolo.

```
    {: RESULT = new ArrayTypeExpression(aleft,t); :} ;

typeplus ::=
    type:t
    {: RESULT = t; :}
  | VOID:v
    {: RESULT = new VoidTypeExpression(vleft); :} ;
```

Il metodo statico `Symbol.mk()` trasforma il valore lessicale `id` di un identificatore `ID` (cioè il suo nome, visto come stringa) in un oggetto della classe `symbol/Symbol.java` mostrata in Figura 3.25. Tale classe è molto simile a `java.lang.String`, con l'unica differenza che non possono esistere due `symbol.Symbol` che rappresentano lo stesso identificatore. Questo è ottenuto costringendo il programmatore a creare oggetti tramite un metodo statico `mk` che tiene traccia degli identificatori già creati ed evita di creare doppioni. Avevamo già osservato infatti che l'albero di sintassi astratta usa gli stessi nodi per diverse occorrenze dello stesso identificatore (Figura 3.1). Il vantaggio di non avere due oggetti diversi per lo stesso identificatore sarà evidente in fase di analisi semantica, quando dovremo associare l'uso di un identificare con la sua dichiarazione. Avere lo stesso oggetto in entrambi i punti di programma semplificherà il nostro lavoro (Capitoli 4 e 5).

```
public class Symbol implements Comparable {
  // alcuni simboli usati di frequente
  public static final Symbol THIS = mk("this");
  public static final Symbol OBJECT = mk("Object");
  public static final Symbol STRING = mk("String");
  private String name;   // il nome del simbolo

  private Symbol(String name) { // si noti: e' private!
    this.name = name;
  }

  // possiamo creare simboli solo con questo metodo
  public static Symbol mk(String name) {
    // usa una tabella statica per determinare se il simbolo e' gia'
    // stato creato e in tal caso lo ritorna. Altrimenti lo crea,
    // lo aggiunge alla tabella e lo ritorna
  }

  public String toString() { return name; }

  public int compareTo(Object other) {
    if (!(other instanceof Symbol)) return 0;
    return name.compareTo(((Symbol)other).name);
  }
}
```

Figura 3.25: La classe `symbol/Symbol.java` che rappresenta gli identificatori Kitten.

Avendo aggiunto delle azioni semantiche alla grammatica della Sezione 3.2, dobbiamo anche definire il tipo del valore semantico dei terminali e dei non terminali della grammatica. A tal fine modifichiamo come segue le enumerazioni della Sezione 3.2.1:

```
terminal String ID, STRING;
terminal Integer INTEGER;
terminal Float FLOATING;

non terminal TypeExpression type;
non terminal TypeExpression typeplus;
```

Le prime tre dichiarazioni dicono che il valore semantico dei token `ID`, `STRING`, `INTEGER` e `FLOATING` è lo stesso sintetizzato dall'analizzatore lessicale per Kitten come valore lessicale per tali token (Capitolo 2). Le ultime due dichiarazioni indicano che il tipo del valore semantico delle

espressioni di tipo è la superclasse `TypeExpression` di tutte le classi astratte per le espressioni di tipo (Figura 3.24).

3.6.2 Le classi di sintassi astratta per le espressioni e per i leftvalue

La Figura 3.26 mostra la gerarchia delle classi di sintassi astratta per espressioni e leftvalue. Vogliamo che il non terminale `exp` per le espressioni abbia un valore lessicale che sia sottoclasse di `Expression`. Per cui dichiariamo:

```
non terminal Expression exp;
```

Per *letterale* si intende una rappresentazione sintattica di un valore. Le classi astratte per i letterali sono create con le produzioni:

```
exp ::=
   INTEGER:i
   {: RESULT = new IntLiteral(ileft,i.intValue()); :}
 | FLOATING:f
   {: RESULT = new FloatLiteral(fleft,f.floatValue()) ; :}
 | STRING:s
   {: RESULT = new StringLiteral(sleft,s); :}
```

Ricordiamo che questi sono i soli tre token che abbiano un valore lessicale associato, oltre ad `ID`.

Le classi di sintassi astratta per i leftvalue sono sottoclassi di `Expression`, il che è sensato essendo i leftvalue dei casi particolari di espressioni. Tali classi di sintassi astratta sono create dalle produzioni:

```
lvalue ::=
   ID:id
   {: RESULT = new Variable(idleft,Symbol.mk(id)); :}
 | exp:receiver DOT:d ID:field
   {: RESULT = new FieldAccess(dleft,receiver,Symbol.mk(field)); :}
 | exp:array LBRACK:b exp:index RBRACK
   {: RESULT = new ArrayAccess(bleft,array,index); :} ;
```

La Figura 3.26 mostra la complessità della gerarchia delle classi di sintassi astratta per le espressioni che sono operatori binari. Tali espressioni sono in primo luogo divise in *aritmetiche* (`ArithmeticBinOp`), *booleane* (`BooleanBinOp`) e *di confronto* (`ComparisonBinOp`). Queste ultime sono a loro volta divise in operazioni di confronto che possono operare su qualsiasi tipo di valore, come l'uguaglianza e la disuguaglianza, e in operazioni di confronto che operano solo su numeri (interi o in virgola mobile), incluse nella classe `NumericalComparisonBinOp`.

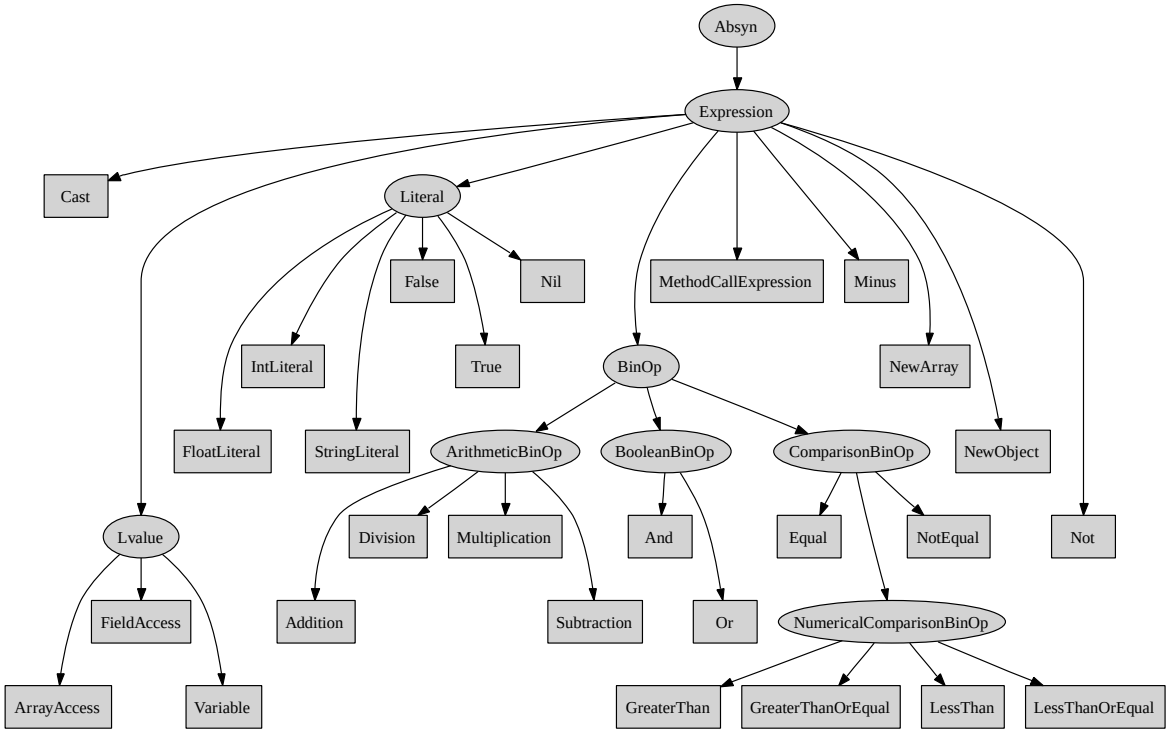


Figura 3.26: La struttura gerarchica delle classi di sintassi astratta per espressioni e leftvalue. Le classi astratte sono indicate con un ovale, quelle concrete con un rettangolo.

3.6.3 Le classi di sintassi astratta per i comandi

La Figura 3.27 mostra le classi di sintassi astratta per i comandi. Si tratta di una gerarchia relativamente semplice. La classe `LocalDeclaration` è utilizzata per rappresentare la dichiarazione di una variabile. La classe `Skip` è usata per rappresentare un comando vuoto, come per esempio il corpo `{}` del costruttore della classe in Figura 1.2. La classe `IfThenElse` è utilizzata per rappresentare sia il condizionale semplice che quello composto, cioè munito del ramo `else`. Questo è evidente osservando le azioni semantiche per tale comando:

```
command ::=
   IF:i LPAREN exp:condition RPAREN THEN command:then
    {: RESULT = new IfThenElse(ileft,condition,then); :}
 | IF:i LPAREN exp:condition RPAREN THEN command:then ELSE command:else
    {: RESULT = new IfThenElse(ileft,condition,then,else); :}
```

Il costruttore a soli tre argomenti della classe `absyn/IfThenElse.java` è definito in modo da chiamare quello a quattro argomenti passando come quarto argomento un ramo `else` vuoto, cioè un'oggetto creato come `new Skip(pos)`. In questo modo d'ora in poi possiamo sempre assumere che i condizionali abbiano sempre un ramo `else`.

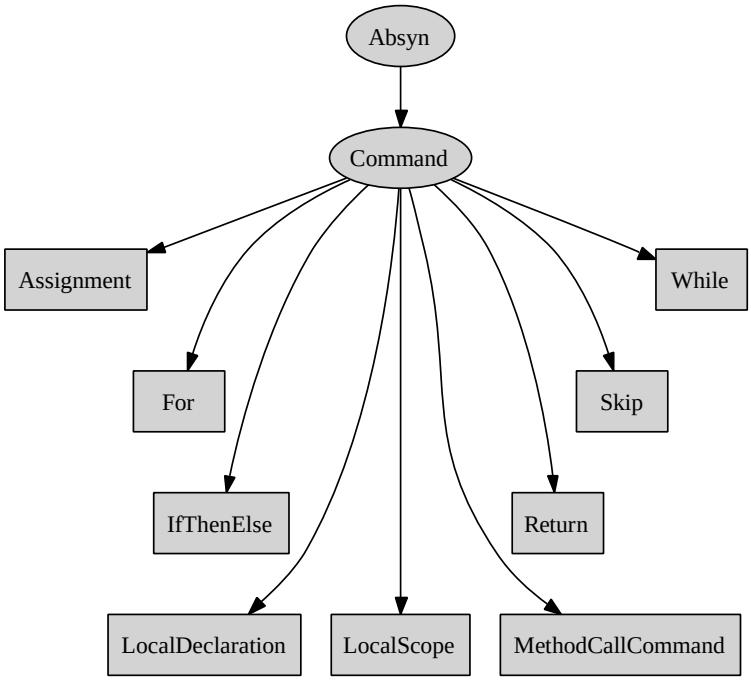


Figura 3.27: La struttura gerarchica delle classi di sintassi astratta per i comandi. Le classi astratte sono indicate con un ovale, quelle concrete con un rettangolo.

Le classi astratte per i comandi hanno un campo `next` che lega in sequenza comandi contigui. Il costruttore di `absyn/Command.java` annulla tale campo, che deve essere settato in seguito quando si riconoscono due comandi contigui. Questo è effettuato tramite il metodo `link()` della classe `absyn/Command.java`, utilizzato nelle produzioni per gli statements:

```
statements ::=
   command:cmd
   {: RESULT = cmd; :}
 | command:cmd SEMICOLON statements:next
   {: cmd.link(next); RESULT = cmd; :} ;
```

Anche per i comandi e gli statement dobbiamo dichiarare il tipo del loro valore semantico, che è la superclasse di tutte le classi di sintassi astratta per i comandi:

```
non terminal Command statements;
non terminal Command command;
```

3.6.4 Le classi di sintassi astratta per le classi Kitten

La Figura 3.28 mostra le classi di sintassi astratta utilizzate per rappresentare la sintassi delle classi Kitten. Una classe Kitten è rappresentata da un oggetto di classe `ClassDefinition` al cui

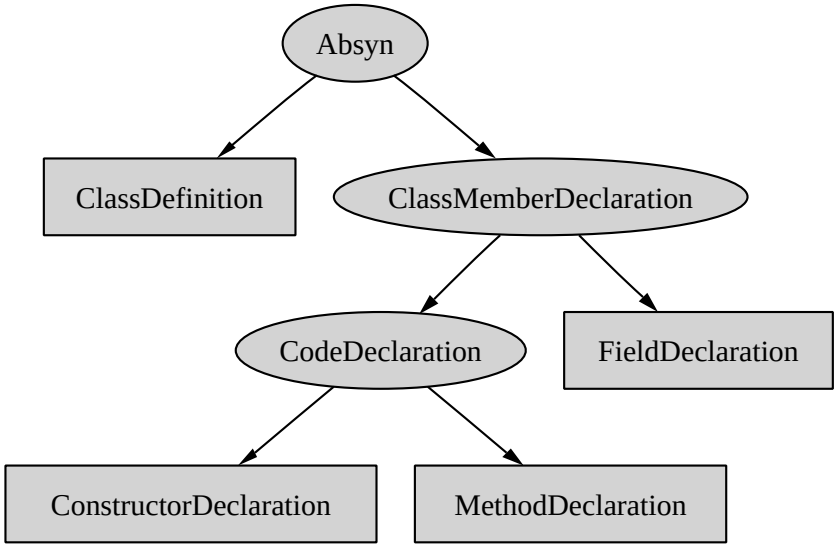


Figura 3.28: La struttura gerarchica delle classi di sintassi astratta usate per rappresentare le classi Kitten. Le classi astratte sono indicate con un ovale, quelle concrete con un rettangolo.

interno si trova una lista di `ClassMemberDeclaration`. Ciascuna di tali dichiarazioni dichiara un membro della classe, che può essere la dichiarazione di un campo, di un costruttore o di un metodo.

Le produzioni che istanziano queste classi di sintassi astratta sono le seguenti (si confronti con la Sezione 3.2.6):

```
class ::=
   CLASS:c ID:name LBRACE class_members:declarations RBRACE
    {: RESULT = new ClassDefinition
      (cleft,Symbol.mk(name),Symbol.OBJECT,declarations); :}
 | CLASS:c ID:name EXTENDS ID:superclass
     LBRACE class_members:declarations RBRACE
    {: RESULT = new ClassDefinition
      (cleft,Symbol.mk(name),Symbol.mk(superclass),declarations); :} ;

class_members ::=
    {: RESULT = null; :}
 | FIELD:f type:t ID:name class_members:next
    {: RESULT = new FieldDeclaration(fleft,t,sym(name),next); :}
 | CONSTRUCTOR:c LPAREN formals:formals RPAREN command:body
     class_members:next
    {: RESULT = new ConstructorDeclaration(cleft,formals,body,next); :}
 | METHOD:m typeplus:returnType ID:name LPAREN formals:formals RPAREN
```

```
 command:body class_members:next
{: RESULT = new MethodDeclaration
  (mleft,returnType,sym(name),formals,body,next); :} ;
```

Si noti che nel caso in cui la superclasse di una classe Kitten non è specificata si assume che essa sia `Object`, in modo che possiamo sempre assumere che una classe Kitten abbia specificata la sua superclasse.

I tipi dei non terminali sono dichiarati come:

```
non terminal ClassDefinition         class;
non terminal ClassMemberDeclaration  class_members;
```

3.6.5 Un riassunto delle classi di sintassi astratta di Kitten

In Figura 3.29 riportiamo un elenco riassuntivo delle classi di sintassi astratta di Kitten. Per ogni classe riportiamo il costruttore, che dà anche informazione sul contenuto degli oggetti di tale classe. Per esempio, la notazione:

```
Addition(Expression left, Expression right)
```

indica che la classe di sintassi astratta `absyn/Addition.java` ha due campi, `left` e `right`, di tipo `Expression`. Il suo costruttore ha in effetti *tre* parametri: oltre ai due riportati, è sottointeso anche un primo parametro `pos` di tipo `int` che indica la posizione del file sorgente in cui segnalare un errore in fase di analisi semantica se qualcosa non torna su questa parte di sintassi. Per semplicità, in Figura 3.29 non abbiamo riportato tale ulteriore parametro.

La Figura 3.29 è compatibile con l'albero di sintassi astratta in Figura 3.1. Per esempio, il nodo etichettato con `Not` in Figura 3.1 rappresenta un oggetto di classe `Not` il cui costruttore, in Figura 3.29, ha intestazione `Not(Expression expression)`. L'espressione negata è infatti legata in Figura 3.1 tramite un arco etichettato con `expression` a un nodo di tipo `FieldAccess`, che è sottoclasse di `Expression` (Figura 3.26). Tale nodo di tipo `FieldAccess` è poi legato, in Figura 3.1, tramite due archi etichettati con `receiver` e `name`, a due nodi di tipo, rispettivamente, `Variable` e `Symbol` (rettangolare). Questo rispetta il costruttore di `FieldAccess` in Figura 3.29, che è `FieldAccess(Expression receiver, Symbol name)`. Si noti che `Variable` è una sottoclasse di `Expression` (Figura 3.26).

Ricordiamo che ulteriori informazioni sulle classi di sintassi astratta sono disponibili con la distribuzione di Kitten, sotto forma di documentazione JavaDoc.

```
Absyn()
Addition(Expression left, Expression right)
And(Expression left, Expression right)
ArithmeticBinOp(Expression left, Expression right)
ArrayAccess(Expression array, Expression index)
ArrayTypeExpression(TypeExpression elementsType)
Assignment(Lvalue lvalue, Expression rvalue)
BinOp(Expression left, Expression right)
BooleanBinOp(Expression left, Expression right)
BooleanTypeExpression()
Cast(TypeExpression type, Expression expression)
ClassDefinition(Symbol name, Symbol superclassName, ClassMemberDeclaration declaration)
ClassMemberDeclaration(ClassMemberDeclaration next)
ClassTypeExpression(Symbol name)
CodeDeclaration(FormalParameters formals, Command body, ClassMemberDeclaration next)
Command()
ComparisonBinOp(Expression left, Expression right)
ConstructorDeclaration(FormalParameters formals, Command body, ClassMemberDeclaration next)
Division(Expression left, Expression right)
Equal(Expression left, Expression right)
Expression()
ExpressionSeq(Expression head, ExpressionSeq tail)
False()
FieldAccess(Expression receiver, Symbol name)
FieldDeclaration(TypeExpression type, Symbol name, ClassMemberDeclaration next)
FloatLiteral(float value)
FloatTypeExpression()
For(Command initialisation, Expression condition, Command update, Command body)
FormalParameters(TypeExpression type, Symbol name, FormalParameters next)
GreaterThan(Expression left, Expression right)
GreaterThanOrEqual(Expression left, Expression right)
IfThenElse(Expression condition, Command then, Command else)
IntLiteral(float value)
IntTypeExpression()
LessThan(Expression left, Expression right)
LessThanOrEqual(Expression left, Expression right)
Literal()
LocalDeclaration(TypeExpression type, Symbol name, Expression initialiser)
LocalScope(Command body)
Lvalue()
MethodCallCommand(Expression receiver, Symbol name, ExpressionSeq actuals)
MethodCallExpression(Expression receiver, Symbol name, ExpressionSeq actuals)
MethodDeclaration(TypeExpression returnType, Symbol name, FormalParameters formals,
                  Command body, ClassMemberDeclaration next)
Minus(Expression expression)
Multiplication(Expression left, Expression right)
NewArray(TypeExpression elementsType, Expression size)
NewObject(Symbol className, ExpressionSeq actuals)
Nil()
Not(Expression expression)
NotEqual(Expression left, Expression right)
NumericalComparisonBinOp(Expression left, Expression right)
Or(Expression left, Expression right)
Return(Expression returned)
Skip()
StringLiteral(String value)
Subtraction(Expression left, Expression right)
True()
TypeExpression()
Variable(Symbol name)
VoidTypeExpression()
While(Expression condition, Command body)
```

Figura 3.29: Una visione d'insieme delle classi di sintassi astratta del linguaggio Kitten. Le classi in *italico* sono classi astratte, nel senso di Java.

Capitolo 4

Discesa Ricorsiva sulla Sintassi Astratta

In questo capitolo discutiamo l'implentazione di algoritmi che scendono ricorsivamente sull'albero di sintassi astratta generato dall'analizzatore sintattico a partire da un file sorgente. Tali algortmi hanno le funzioni più svariate, da una semplice raccolta di dati statistici sul codice alla verifica dell'uso corretto degli identificatori, dalla determinazione del *codice morto*, cioè di porzioni di codice che sicuramente non vengono raggiunte dal flusso di controllo e possono quindi essere eliminate, alla rappresentazione grafica della sintassi astratta, fino al type-checking del codice, a cui dedicheremo, per la sua complessità, l'intero Capitolo 5 o addirittura alla generazione del codice intermedio, descritta nel Capitolo 6. L'implementazione di questi algoritmi tramite discesa ricorsiva sulla sintassi astratta, piuttosto che tramite azioni semantiche (Capitolo 3), permette di sfruttare la gerarchia delle classi di sintassi astratta per Kitten descritta nel Capitolo 3, definendo l'algoritmo ricorsivo tramite un metodo virtuale ricorsivo delle classi di sintassi astratta.

Si consideri per esempio la grammatica in Figura 3.8. Supponiamo di voler contare il numero di `a` e il numero di `b` di ciascun file sorgente che rispetta le regole della grammatica. Abbiamo già risolto questo problema in Figura 3.19 tramite azioni semantiche. Ma possiamo fare la stessa cosa generando la sintassi astratta tramite le regole in Figura 3.21 e aggiungendo alle classi di sintassi astratta un metodo `count()` che conta il numero di `a` o di `b` contenute nella sintassi concreta rappresentata dall'albero di sintassi astratta. Il codice è mostrato nella Figura 4.1. Le classi `AbstractA`, `EmptyA` e `OneA` sono simili a quelle mostrate nella figura per `B`. La classe `Pair` è la stessa usata nella Sezione 3.5 (una coppia di interi).

Va subito osservato che in Figura 4.1 sono definiti *due* metodi `count()`. Il primo è quello ricorsivo delle classi `EmptyB` e `OneB`; il secondo è quello *non ricorsivo* della classe `AB`. Esiste anche un metodo `count()` ricorsivo dentro `EmptyA` e `OneA` che non è mostrato in figura in quan-

```
public abstract class AbstractB {
  public abstract int count();
}

public class EmptyB extends AbstractB {
  public int count() { return 0; }
}

public class OneB extends AbstractB {
  private AbstractB l;
  public OneB(AbstractB l) { this.l = l; }
  public int count() { return 1 + l.count(); }
}

public class AB {
  private AbstractA a;
  private AbstractB b;
  public AB(AbstractA a, AbstractB b) { this.a = a; this.b = b; }
  public Pair count() {
    return new Pair(a.count(),b.count());
  }
}
```

Figura 4.1: Un metodo ricorsivo che conta il numero di `a` e di `b` nella sintassi astratta generata come in Figura 3.19.

to è simmetrico a quello in `EmptyB` e `OneB`. Il metodo `count()` di `EmptyB` e `OneB` è *a discesa ricorsiva sulla sintassi astratta* poiché la sua chiamata scende ricorsivamente sui componenti della sintassi astratta fino ad arrivare alle foglie, per cui è definito un valore costante. Si faccia attenzione al fatto che il metodo virtuale `count()` è dichiarato `abstract` nella superclasse astratta `AbstractB` e quindi *deve* essere implementato in ognuna delle sue sottoclassi.

Cosa abbiamo guadagnato rispetto all'uso di azioni semantiche come in Figura 3.19? In primo luogo abbiamo lasciato l'analizzatore sintattico alla sua occupazione più specifica, cioè all'analisi sintattica, piuttosto che usarlo per funzioni *improprie* tramite azioni semantiche. In secondo luogo abbiamo *concentrato* delle funzioni della sintassi astratta nelle classi di sintassi astratta stesse, sotto forma di codice Java ricorsivo, che può essere complesso quanto vogliamo. In terzo luogo possiamo utilizzare la gerarchizzazione delle classi di sintassi astratta (come quella discussa per Kitten nella Sezione 3.6) per definire i metodi virtuali solo su alcune superclassi, lasciandoli ereditare a tutte le loro sottoclassi, nei casi in cui essi non differissero da una sottoclasse all'altra. Questo non è possibile tramite azioni semantiche, che anche se fossero uguali devono essere comunque duplicate per ogni produzione della grammatica. Quest'ultimo aspetto sarà chiaro adesso considerando un esempio di discesa ricorsiva sulla sintassi astratta di Kitten.

4.1 Determinazione delle variabili di un'espressione o comando

Espressioni e comandi Kitten possono contenere variabili. Per esempio, le variabili che *occorrono* (sono contenute) nell'espressione Kitten `x + 3 * y` sono `x` e `y`. Le variabili che occorrono nel comando Kitten `x := 12 + a` sono `x` e `a`. Ci proponiamo adesso di definire formalmente e poi di calcolare l'insieme delle variabili che occorrono in un'espressione e poi l'insieme delle variabili che occorrono in un comando.

Cominciamo con le espressioni (Figura 3.26). Definiamo una funzione

$$\mathsf{vars}[\![\_]\!] : \texttt{Expression} \mapsto \wp(\texttt{Symbol})$$

che mappa ciascuna espressione nell'insieme delle sue variabili (*simboli*, Figura 3.25). Alcuni casi della definizione di questa funzione sono particolarmente semplici. Per esempio, una variabile occorre in se stessa:

$$\mathsf{vars}[\![\texttt{Variable(name)}]\!] = \{\texttt{name}\} \tag{4.1}$$

mentre i letterali non contengono variabili:

$$\mathsf{vars}[\![\texttt{Literal()}]\!] = \emptyset\,. \tag{4.2}$$

Si noti che quest'ultima equazione definisce l'insieme delle variabili che occorrono in ogni sottoclasse di `Literal` in Figura 3.26, senza bisogno di ripetere l'equazione per ogni sottoclasse.

Un po' più complesso è il caso del meno unario e della negazione logica, che contenendo ricorsivamente un'altra espressione danno origine a una definizione ricorsiva per $\mathsf{vars}[\![]\!]$ (ma ben fondata perché andiamo verso strutture sintattiche sempre più piccole):

$$\mathsf{vars}[\![\texttt{Minus(expression)}]\!] = \mathsf{vars}[\![\texttt{Not(expression)}]\!] = \mathsf{vars}[\![\texttt{expression}]\!]\,.$$

Un cast e la creazione di un array contengono sia un tipo che un'espressione. Il tipo non contribuisce alle variabili, ma solo l'espressione:

$$\begin{aligned}\mathsf{vars}[\![\texttt{Cast(type, expression)}]\!] &= \mathsf{vars}[\![\texttt{expression}]\!]\\ \mathsf{vars}[\![\texttt{NewArray(elementsType, size)}]\!] &= \mathsf{vars}[\![\texttt{size}]\!]\,.\end{aligned}$$

Similmente per l'accesso a un campo, in cui l'identificatore fa riferimento a un campo e non a una variabile e quindi non è di nostro interesse in questo contesto:

$$\mathsf{vars}[\![\texttt{FieldAccess(receiver, name)}]\!] = \mathsf{vars}[\![\texttt{receiver}]\!]\,. \tag{4.3}$$

La definizione per *tutti* gli operatori binari può venire data come:

$$\mathsf{vars}[\![\texttt{BinOp(left, right)}]\!] = \mathsf{vars}[\![\texttt{left}]\!] \cup \mathsf{vars}[\![\texttt{right}]\!] \tag{4.4}$$

e similmente quella per l'accesso a un elemento di un array (sia l'array che l'indice dell'elemento sono espressioni):

$$\mathsf{vars}[\![\texttt{ArrayAccess}(\texttt{array}, \texttt{index})]\!] = \mathsf{vars}[\![\texttt{array}]\!] \cup \mathsf{vars}[\![\texttt{index}]\!] \,.$$

Rimangono le chiamate di metodo e la creazione di un oggetto, che contengono una lista di espressioni (parametri). Definiamo l'insieme delle variabili che occorrono in una lista di espressioni come l'unione delle variabili che occorrono in ciascuna espressione:

$$\begin{aligned} \mathsf{vars}[\![\texttt{MethodCallExpression}(\texttt{receiver}, \texttt{name}, \texttt{actuals})]\!] &= \mathsf{vars}[\![\texttt{receiver}]\!] \\ &\quad \cup\ \mathsf{vars}[\![\texttt{actuals}]\!] \\ \mathsf{vars}[\![\texttt{NewObject}(\texttt{className}, \texttt{actuals})]\!] &= \mathsf{vars}[\![\texttt{actuals}]\!] \end{aligned}$$

dove $\mathsf{vars}[\![\_]\!] : \texttt{ExpressionSeq} \mapsto \wp(\texttt{Symbol})$ è definito come

$$\begin{aligned} \mathsf{vars}[\![\texttt{null}]\!] &= \emptyset \\ \mathsf{vars}[\![\texttt{ExpressionSeq}(\texttt{head}, \texttt{tail})]\!] &= \mathsf{vars}[\![\texttt{head}]\!] \cup \mathsf{vars}[\![\texttt{tail}]\!] \,. \end{aligned}$$

Consideriamo adesso i comandi (Figura 3.27). Estendiamo la funzione $\mathsf{vars}[\![\_]\!]$ in modo da potere essere applicata anche ai comandi:

$$\mathsf{vars}[\![\_]\!] : \texttt{Command} \mapsto \wp(\texttt{Symbol}) \,.$$

L'idea è semplicissima: le variabili che occorrono in un comando sono quelle che occorrono in uno qualsiasi dei componenti del comando (espressioni o sotto-comandi). Inoltre una sequenza di comandi separati da punto e virgola contiene l'unione delle variabili contenute in ciascun comando:

$$\begin{aligned} \mathsf{vars}[\![\texttt{Assignment}(\texttt{lvalue}, \texttt{rvalue})]\!] &= \mathsf{vars}[\![\texttt{lvalue}]\!] \cup \mathsf{vars}[\![\texttt{rvalue}]\!] \\ \mathsf{vars}[\![\texttt{For}(\texttt{initialisation}, \texttt{condition}, \texttt{update}, \texttt{body})]\!] &= \mathsf{vars}[\![\texttt{initialisation}]\!] \\ &\quad \cup\ \mathsf{vars}[\![\texttt{condition}]\!] \\ &\quad \cup\ \mathsf{vars}[\![\texttt{update}]\!] \\ &\quad \cup\ \mathsf{vars}[\![\texttt{body}]\!] \\ \mathsf{vars}[\![\texttt{IfThenElse}(\texttt{condition}, \texttt{then}, \texttt{else})]\!] &= \mathsf{vars}[\![\texttt{condition}]\!] \\ &\quad \cup\ \mathsf{vars}[\![\texttt{then}]\!] \cup \mathsf{vars}[\![\texttt{else}]\!] \qquad (4.5) \\ \mathsf{vars}[\![\texttt{LocalDeclaration}(\texttt{type}, \texttt{name}, \texttt{initialiser})]\!] &= \{\texttt{name}\} \cup \mathsf{vars}[\![\texttt{initialiser}]\!] \\ \mathsf{vars}[\![\texttt{LocalScope}(\texttt{body})]\!] &= \mathsf{vars}[\![\texttt{body}]\!] \\ \mathsf{vars}[\![\texttt{MethodCallCommand}(\texttt{receiver}, \texttt{name}, \texttt{actuals})]\!] &= \mathsf{vars}[\![\texttt{receiver}]\!] \\ &\quad \cup\ \mathsf{vars}[\![\texttt{actuals}]\!] \\ \mathsf{vars}[\![\texttt{Return}(\texttt{returned})]\!] &= \mathsf{vars}[\![\texttt{returned}]\!] \\ \mathsf{vars}[\![\texttt{Skip}()]\!] &= \emptyset \\ \mathsf{vars}[\![\texttt{While}(\texttt{condition}, \texttt{body})]\!] &= \mathsf{vars}[\![\texttt{condition}]\!] \cup \mathsf{vars}[\![\texttt{body}]\!] \\ \mathsf{vars}[\![com_1; \cdots ; com_n]\!] &= \cup_{i=1}^{n} \mathsf{vars}[\![com_i]\!] \,. \qquad (4.6) \end{aligned}$$

Consideriamo adesso l'implementazione in Java della funzione vars[[]] che abbiamo appena finito di definire. L'idea è di aggiungere un metodo `vars()` alle classi di sintassi astratta per espressioni e comandi. Cominciamo dichiarandoli `abstract` all'interno delle superclassi astratte di tutte le espressioni e di tutti i comandi:

```
public class Expression extends Absyn {
  ....
  public abstract HashSet<Symbol> vars();
}

public class Command extends Absyn {
  ...
  public abstract HashSet<Symbol> vars();
}
```

A questo punto abbiamo l'obbligo di istanziare tale metodo in tutte le sottoclassi di `Expression` (Figura 3.26) e in tutte le sottoclassi di `Command` (Figura 3.27). Il compito sembra gravoso ma è in realtà semplificato dalla gerachia delle classi. Per esempio, l'Equazione (4.1) viene implementata modificando la classe `absyn/Variable.java`:

```
public class Variable extends Lvalue {
  ...
  public HashSet<Symbol> vars() {
    HashSet<Symbol> result = new HashSet<Symbol>();
    result.add(name);
    return result;
  }
}
```

e l'Equazione (4.2) viene implementata *per tutti i letterali* modificando soltanto la loro superclasse `absyn/Literal.java`:

```
public class Literal extends Expression {
  ...
  public final HashSet<Symbol> vars() {
    return new HashSet<Symbol>();  // insieme vuoto
  }
}
```

Si noti che il metodo `vars()` di `absyn/Literal.java` è lasciato ereditare a tutte le sue sottoclassi, che quindi non ne ricevono una definizione esplicita.

Definizioni ricorsive diventano implementazioni ricorsive del metodo `vars()`. Per esempio, l'Equazione (4.3) è implementata modificando coma segue la classe `absyn/FieldAccess.java`:

```
public class FieldAccess extends Lvalue {
  ...
  public HashSet<Symbol> vars() {
    return receiver.vars();
  }
}
```

Il metodo `vars()` *per tutti gli operatori binari* è definito modificando, in accordo con l'Equazione (4.4), la classe `absyn/BinOp.java`:

```
public class BinOp extends Expression {
  ...
  public final HashSet<Symbol> vars() {
    HashSet<Symbol> result = new HashSet<Symbol>();
    result.addAll(left.vars());
    result.addAll(right.vars());
    return result;
  }
}
```

Ci si abitui a definire i metodi a discesa ricorsiva come nell'esempio precedente, in cui cioè il risultato delle chiamate ricorsive non è modificato per costruire il risultato complessivo. Questo approccio, che chiamiamo *funzionale*, è preferibile a un approccio *distruttivo* del tipo `HashSet<Symbol> result = left.vars(); result.addAll(right.vars());` in cui l'insieme proveniente dalla sotto-espressione `left` è modificato aggiungendovi i simboli provenienti dalla sotto-espressione `right`. Il motivo per cui preferiamo un approccio funzionale è che quest'ultimo lascia immutati i risultati intermedi delle chiamate ricorsive. Questi risultati potrebbero essere salvati e poi riutilizzati in futuro. In alcuni casi un approccio distruttivo può introdurre veri e propri errori di programmazione a causa di una complessa condivisione di strutture dati che lo studente non riesce a dominare.

Consideriamo adesso i comandi. Il metodo `vars()` per il comando condizionale andrebbe scritto modificando, in accordo con l'Equazione (4.5), la classe `absyn/IfThenElse.java`. Va però osservato che i comandi sono legati a lista, per cui dopo aver calcolato le variabili che occorrono nel comando condizionale bisogna aggiungere quelle che occorrono nei comandi ad esso legati tramite il campo `next` (in accordo con l'Equazione (4.6)). Conseguentemente otteniamo:

```
public class IfThenElse extends Command {
  ...
  public HashSet<Symbol> vars() {
    HashSet<Symbol> result = new HashSet<Symbol>();
    result.addAll(condition.vars());
    result.addAll(then.vars());
    result.addAll(else.vars());
```

```
    if (next != null) result.addAll(next.vars());
    return result;
  }
}
```

Si osservi che delle quattro chiamate a `vars()`, la prima avviene sulle espressioni mentre le tre successive sui comandi.

L'aggiunta delle variabili usate nel comando `next` (se esso non è `null`) non è specifica al comando condizionale ma va fatta per tutti i comandi. Conseguentemente è un po' più semplice effettuare tale aggiunta per default e non considerarla ogni volta per ciascun comando. Questo ha inoltre l'effetto di far coincidere l'implementazione con la specifica formale data come nell'Equazione (4.5). Infine un ragionamento di default evita di dimenticare la linea

```
if (next != null) result.addAll(next.vars());
```

in eventuali altri comandi che verranno aggiunti in futuro a Kitten. Per ottenere questo meccanismo di default basta rimodificare la classe `absyn/Command.java` come segue:

```
public class Command extends Absyn {
  ...
  public final HashSet<Symbol> vars() {
    HashSet<Symbol> result = vars$0();
    if (next != null) result.addAll(next.vars());
    return result;
  }

  protected abstract HashSet<Symbol> vars$0();
}
```

Abbiamo cioè reso `final` il metodo `vars()`, il quale adesso per default chiama un nuovo metodo `vars$0()` dei comandi e poi aggiunge le variabili usate da `next`, se esso non è `null`. Si noti che il metodo ausiliario `vars$0()` è `protected` in modo da nasconderne la visibilità all'interno del package `absyn` e delle sottoclassi di `absyn/Command.java`. A questo punto la classe `absyn/IfThenElse.java` viene modificata come segue:

```
public class IfThenElse extends Command {
  ...
  public HashSet<Symbol> vars$0() {
    HashSet<Symbol> result = new HashSet<Symbol>();
    result.addAll(condition.vars());
    result.addAll(then.vars());
    result.addAll(else.vars());
    return result;
  }
}
```

Si noti che essa non contiene più il metodo `vars()` (e non potrebbe mai contenerlo perché l'abbiamo dichiarato `final` in `absyn/Command.java` al fine di evitarne ridefinizioni).

4.2 Determinazione delle variabili dichiarate ma non usate

Kitten ammette di dichiarare variabili che poi non vengono usate all'interno del loro scope di visibilità. È comunque ovvio che la dichiarazione di una variabili non usata è inutile e può essere eliminata dal programma senza cambiare la semantica di quest'ultimo, purché l'inizializzatore della variabile non contenga side-effect e la sua valutazione termini sempre. Comunque sia, la dichiarazione di una variabile non usata è spesso considerata erronea o sospetta (*warning*) in molti altri linguaggi di programmazione.

Supponiamo di volere aggiungere a Kitten la possibilità di controllare che il corpo di metodi e costruttori non dichiari mai variabili non utilizzate. Facciamo l'ipotesi semplificativa che ogni dichiarazione in un metodo o costruttore introduca una variabile diversa, ovvero che non ci siano *sinonimi*. Questa ipotesi può essere rimossa ridenominando opportunamente le variabili del programma. Dal momento che il corpo di un metodo o costruttore è un comando c (Sezione 3.2.6), possiamo determinare l'insieme delle variabili dichiarate ma non usate in c come la sottrazione dell'insieme delle variabili usate in c da quello delle variabili dichiarate in c:

$$\mathsf{declared}[\![c]\!] \setminus \mathsf{used}[\![c]\!] \ .$$

Si noti che consideriamo solo le variabili dichiarate in c e non i parametri del metodo o costruttore e neppure il parametro implicito `this`. Questo implica che accettiamo l'idea che i parametri, incluso il parametro implicito `this`, possano non venire usati all'interno del metodo o costruttore, senza che questo comporti la segnalazione di un warning al programmatore[1].

Sappiamo già calcolare l'insieme delle variabili che *occorrono* in un comando (Sezione 4.1). L'insieme delle variabili *usate* in un comando coincide con l'insieme calcolato dalla funzione $\mathsf{vars}[\![\,]\!]$ se non fosse che la dichiarazione di una variabile non deve essere considerata come un uso della stessa. Conseguentemente la definizione di $\mathsf{used}[\![\,]\!]$ è identica a quella di $\mathsf{vars}[\![\,]\!]$ data nella Sezione 4.1 con l'unica differenza che

$$\mathsf{used}[\![\texttt{LocalDeclaration}(\texttt{type}, \texttt{name}, \texttt{initialiser})]\!] = \mathsf{used}[\![\texttt{initialiser}]\!] \ .$$

Invece dobbiamo ancora definire l'insieme delle variabili *dichiarate* in un comando. Lo facciamo dicendo che normalmente un comando non dichiara alcuna variabile: $\mathsf{declared}[\![c]\!] = \emptyset$, ma la dichiarazione di variabile fa ovviamente eccezione alla regola generale:

$$\mathsf{declared}[\![\texttt{LocalDeclaration}(\texttt{type}, \texttt{name}, \texttt{initialiser})]\!] = \{\texttt{name}\} \ .$$

Infine, l'insieme delle variabili dichiarate in una sequenza di comandi separati da punto e virgola è l'unione degli insiemi delle variabili dichiarate in ciascun comando della sequenza:

$$\mathsf{declared}[\![com_1 ; \cdots ; com_n]\!] = \cup_{i=1}^{n} \mathsf{declared}[\![com_i]\!] \ .$$

L'implementazione in Java della funzione $\mathsf{used}[\![\,]\!]$ è quasi identica all'implementazione della funzione $\mathsf{vars}[\![\,]\!]$, vista nella Sezione 4.1. L'unica differenza è per la dichiarazione di variabile.

[1] Non utilizzare i parametri è estremamente usuale in un contesto di programmazione a oggetti in cui i metodi di una classe sono spesso ottenuti per specializzazione da quelli della superclasse. Questa ipotesi potrebbe quindi essere migliorata per i soli metodi privati di una classe, ma Kitten non ammette metodi privati.

L'implementazione della funzione declared⟦⟧ invece è interessante poiché permette di sfruttare la gerarchia delle classi in Figura 3.27. Definiamo infatti un comportamento di default per il metodo `declared()` che aggiungiamo ai comandi:

```
public class Command extends Absyn {
  ...
  public HashSet<Symbol> declared() {
    if (next != null) return next.declared();
    else return new HashSet<Symbol>();    // un insieme vuoto
  }
}
```

Quindi definiamo l'unica eccezione in `absyn/LocalDeclaration.java` ridefinendovi il metodo `declared()`:

```
public class LocalDeclaration extends Command {
  ...
  public HashSet<Symbol> declared() {
    HashSet<Symbol> result = new HashSet<Symbol>();
    result.add(name);
    if (getNext() != null) result.addAll(getNext().declared());
    return result;
  }
}
```

In `absyn/LocalDeclaration.java` dobbiamo usare il metodo `getNext()` per accedere al campo `next` poiché quest'ultimo è locale ad `absyn/Command.java`.

La tecnica descritta per determinare l'insieme delle variabili dichiarate ma non usate è abbastanza rudimentale. Abbiamo già detto che è necessario supporre che non ci siano variabili con lo stesso nome dichiarate all'interno dello stesso metodo o costruttore. Inoltre la tecnica restituisce solo i nomi delle variabili che sono state dichiarate ma non usate. Conseguemente possiamo dare un errore solo alla fine del metodo piuttosto che nel punto esatto in cui la variabile è stata dichiarata (si potrebbe ridiscendere sulla sintassi astratta per segnalare l'errore al posto giusto...). Infine osserviamo che questo metodo non capisce se una variabile è usata solo all'interno di una parte di codice che non verrà mai eseguita, nel qual caso essa non è *realmente* usata. E non si potrà mai risolvere del tutto questo problema poiché l'eseguibilità di porzioni di codice è una proprietà indecibile. Come per tutte le proprietà *interessanti* di programmi, dovremo accontentarci di approssimazioni per le informazioni che cerchiamo, purché esse siano approssimazioni *corrette*. Per esempio, in questa sezione la correttezza significa che se una variabile sta in declared⟦c⟧ \ vars⟦c⟧ allora sicuramente essa è stata dichiarata ma non usata in c. Non abbiamo nessuna garanzia del contrario: basta considerare il comando `int y := 4; if (x = x) then {} else x := y` in cui `y` è solo apparentemente usata, poiché il ramo `else` del condizionale non verrà mai eseguito. Ovviamente questo esempio può essere complicato a piacere, essendo il problema indecibile.

4.3 Determinazione del codice morto

Un comando è detto *codice morto* in un programma se esso non verrà mai eseguito, indipendentemente dall'input fornito al programma. Conseguentemente, tale comando è *inutile* e potrebbe essere eliminato. Determinare se un comando è codice morto è un problema indecidibile. Ci accontenteremo quindi di una approssimazione *corretta* dell'insieme del codice morto, nel senso che se riusciremo a determinare che un comando è codice morto allora esso lo è realmente, ma il viceversa in genere non sarà vero: ci sarà del codice morto che sfuggirà alla nostra tecnica di ricerca.

Un esempio di codice morto è l'ultimo comando del seguente metodo:

```
method int fib(int i) {
  if (i < 2) then return 1
  else return this.fib(i - 1) + this.fib(i - 2);

  "ciao".output()
}
```

Il motivo è che la linea `"ciao".output()` non può mai essere raggiunta. Evitare la compilazione di programmi che contengono codice morto può sembrare eccessivamente restrittivo, ma è invece spesso importante poiché costringe il programmatore a ragionare sulla struttura di controllo del codice. Molto spesso, infatti, la presenza di codice morto è un sintomo di un bug in un programma. Si noti che se ci fosse un ulteriore comando nel corpo del metodo precedente, subito dopo il comando `"ciao".output()`, allora preferiremmo non segnalare che esso è codice morto, benché effettivamente lo sia. Questo al fine di non confondere il programmatore con errori a cascata che finiscono per essere poco focalizzati sul punto problematico del codice.

Un problema imparentato a quello del codice morto è quello del codice che non termina necessariamente con un `return`. Questo è importante nel caso di metodi che non ritornano `void`, come per esempio

```
method int fib(int i)
  if (i < 2) then return 1
```

Poiché non è specificato cosa deve essere ritornato nel caso in cui il parametro `i` è maggiore o uguale a 2, il precedente metodo deve essere rifiutato come incorretto e non compilato in codice eseguibile. In altre parole, per i metodi che non ritornano `void` pretendiamo che qualsiasi percorso che porta a concludere l'esecuzione del metodo termini con un'istruzione `return`.

L'approccio che seguiamo per determinare il codice morto e al contempo garantire che il corpo di metodi non `void` termini sempre con un `return` è ancora una volta la discesa ricorsiva sulla sintassi astratta. Questa volta però rappresentiamo il nostro algoritmo tramite *regole di inferenza* piuttosto che tramite definizioni denotazionali. Sebbene le due tecniche siano largamente interscambiabili, è bene conoscerle entrambe poiché le definizioni denotazionali sono spesso più compatte mentre le regole di inferenza permettono di esprimere meglio delle condizioni di errore, come nel caso che stiamo per considerare.

Partiamo da una sequenza di comandi $com_1; com_2$. In che situazione siamo certi che com_2 non è mai eseguito? Sicuramente quando l'esecuzione di com_1 termina sempre e comunque con un comando `return` che fa uscire dal metodo in cui siamo. Se quindi avessimo un *giudizio* $c \vdash^{\mathsf{cdc}} b$, dove cdc sta per *Check for Dead-Code* e in cui il booleano b è vero quando il singolo comando c termina sempre e comunque eseguendo un comando `return` che fa uscire dal metodo in cui siamo, allora potremmo estendere $\vdash^{\mathsf{cdc}}$ a coppie di comandi con la regola:

$$\frac{\text{se } com_1 \vdash^{\mathsf{cdc}} \mathit{true} \text{ segnala un warning} \qquad com_2 \vdash^{\mathsf{cdc}} b}{com_1; com_2 \vdash^{\mathsf{cdc}} b}$$

facilmente estendibile poi ulteriormente a sequenze di comandi arbitrariamente lunghe:

$$\frac{\text{se } com_1 \vdash^{\mathsf{cdc}} \mathit{true} \text{ segnala un warning} \qquad com_2; \cdots; com_n \vdash^{\mathsf{cdc}} b}{com_1; com_2; \cdots; com_n \vdash^{\mathsf{cdc}} b} \tag{4.7}$$

Il predicato $\vdash^{\mathsf{cdc}}$ è utile anche per garantire che il corpo c di un metodo non `void` termini sempre con un comando `return`: basta richiedere che $c \vdash^{\mathsf{cdc}} \mathit{true}$.

Una regola come l'Equazione (4.7) va vista come un'implicazione dall'alto (premesse) in basso (conseguenza). In alto ammettiamo di aggiungere delle annotazioni utili a segnalare warning o errori, ma che non sono premesse. Quindi le regole precedenti hanno ciascuna una sola premessa. Una regola è immediatamente trasformata in del codice Java che la implementa, in cui le premesse sono il corpo dell'implementazione. Vedremo fra un attimo degli esempi. Va detto che questa proprietà *operazionale* è sicuramente un vantaggio delle regole di inferenza rispetto a una specifica denotazionale.

Si noti che l'Equazione (4.7) determina se l'*ultimo* comando della sequenza termina sempre e comunque con un `return` che fa uscire dal metodo in cui siamo, al fine di evitare warning a cascata e di focalizzare l'attenzione del programmatore sul punto in cui l'errore è originato.

Vediamo adesso come definiamo il predicato $\vdash^{\mathsf{cdc}}$ sui singoli comandi della Figura 3.27. Un comando `return` termina sempre e comunque con se stesso e fa uscire dal metodo in cui occorre, per cui:

$$\frac{}{\texttt{Return(returned)} \vdash^{\mathsf{cdc}} \mathit{true}}$$

Questa regola non ha premesse ed è chiamata *assioma* o *fatto*. La sua conseguenza è *sempre* vera.

Il caso del condizionale è molto interessante. Affinché la sua esecuzione termini sempre e comunque con un comando `return` che fa uscire dal metodo in cui siamo, questo deve essere vero per *entrambi* i suoi rami `then` ed `else`. Infatti non siamo abbastanza *raffinati* da scoprire che uno dei due rami magari non è mai eseguito. Conseguentemente definiamo:

$$\frac{\texttt{then} \vdash^{\mathsf{cdc}} b_1 \quad \texttt{else} \vdash^{\mathsf{cdc}} b_2}{\texttt{IfThenElse(condition, then, else)} \vdash^{\mathsf{cdc}} b_1 \wedge b_2} \tag{4.8}$$

I cicli sono particolarmente subdoli. Sembrerebbe infatti a prima vista che se il corpo di un `while` termina sempre e comunque con un `return` che fa uscire dal metodo in cui siamo allora lo stesso è vero per l'intero comando `while`. Ma questo sarebbe vero solo se fossimo capaci di

dimostrare che il corpo del `while` viene eseguito *almeno una volta*, perché altrimenti l'esecuzione continuerebbe con l'istruzione successiva, senza incontrare alcun `return`. Dal momento che non siamo abbastanza raffinati da sapere se un ciclo viene eseguito almeno una volta, siamo costretti, per sicurezza, a dire che i cicli non terminano mai con un `return` che fa uscire dal metodo in cui siamo:

$$\frac{\texttt{body} \vdash^{\text{cdc}} b}{\texttt{While(condition, body)} \vdash^{\text{cdc}} \mathit{false}} \tag{4.9}$$

Perché abbiamo comunque richiesto come premessa la discesa ricorsiva sul corpo del `while`? Poiché altrimenti dell'eventuale codice morto all'interno di `body` non sarebbe stato scoperto. Si consideri per esempio il ciclo

```
while (x > 0) {
  x := x - 1;
  return;
  y := y + 1
}
```

in cui il comando `y := y + 1` è codice morto e viene identificato dalla regola (4.9) proprio grazie alla sua premessa.

Il caso del ciclo `for` è simile a quello del ciclo `while` ma va tenuto conto che almeno l'inizializzazione del ciclo viene *sempre* effettuata una volta. Conseguentemente, se essa eseguisse sempre e comunque un comando `return` che fa uscire dal metodo in cui siamo lo stesso accadrebbe per l'intero `for`. Definiamo quindi:

$$\frac{\texttt{initialisation} \vdash^{\text{cdc}} b_1 \quad \text{se } b_1 = \mathit{true} \text{ segnala un warning} \quad \texttt{update} \vdash^{\text{cdc}} b_2 \quad \texttt{body} \vdash^{\text{cdc}} b_3}{\texttt{For(initialisation, condition, update, body)} \vdash^{\text{cdc}} \mathit{false}} \tag{4.10}$$

Si noti che, anche quando $b_1 = \mathit{true}$, il giudizio per l'intero `for` è *false* poiché non vogliamo dare warning a cascata.

Il caso della dichiarazione di uno scope locale è gestito dalla regola

$$\frac{\texttt{body} \vdash^{\text{cdc}} b}{\texttt{LocalScope(body)} \vdash^{\text{cdc}} b}$$

Essa dice che l'esecuzione di uno scope locale termina sempre e comunque con un comando `return` che fa uscire dal metodo in cui siamo se questo avviene per il comando che sta dentro allo scope (se `body` fosse una sequenza di comandi, si considera il suo ultimo comando, per via della regola (4.7)).

I restanti comandi non terminano mai la loro esecuzione con un `return` che fa uscire dal metodo in cui occorrono:

$$\frac{}{\texttt{Skip()} \vdash^{\text{cdc}} \mathit{false}}$$

$$\frac{}{\texttt{Assignment(lvalue, rvalue)} \vdash^{\text{cdc}} \mathit{false}}$$

$$\frac{}{\texttt{LocalDeclaration(type, name, initialiser)} \vdash^{\text{cdc}} \mathit{false}}$$

$$\frac{}{\texttt{MethodCallCommand}(\texttt{receiver}, \texttt{name}, \texttt{actuals}) \vdash^{\text{cdc}} \mathit{false}}$$

L'implementazione del giudizio $\vdash^{\text{cdc}}$ è fatta tramite un metodo `checkForDeadcode$0()` dei comandi, che implementa $\vdash^{\text{cdc}}$ per comandi singoli, e un metodo `checkForDeadcode()`, sempre dei comandi, che lo estende a sequenze di comandi legati dal campo `next`, secondo l'Equazione (4.7). A tal fine modifichiamo `absyn/Command.java` come segue:

```
public class Command extends Absyn {
  ...
  public final boolean checkForDeadcode() {
    boolean stopping = checkForDeadcode$0(checker);
    if (next != null) {
      if (stopping) error("unreachable code after this statement");
      return next.checkForDeadcode(checker);
    }
    else return stopping;
  }

  protected abstract boolean checkForDeadcode$0();
}
```

Il metodo `checkForDeadcode()` applicato a un comando singolo non fa altro che chiamare `checkForDeadcode$0()`; applicato a un comando composto controlla che il primo non termini sempre e comunque con un `return` che fa uscire dal metodo in cui occorre (variabile `stopping`) segnalando un warning in tal caso, quindi prosegue con la chiamata ricorsiva sui comandi che seguono. Il metodo `abstract checkForDeadcode$0()`, come abbiamo già detto, è l'implementazione di $\vdash^{\text{cdc}}$ per un singolo comando. Per esempio, nel caso del condizionale modifichiamo la classe `absyn/IfThenElse.java` definendo:

```
public class IfThenElse extends Command {
  ...
  protected boolean checkForDeadcode$0() {
    return then.checkForDeadcode() && else.checkForDeadcode();
  }
}
```

in accordo con l'Equazione (4.8). Si noti che la chiamata ricorsiva è stata fatta tramite il metodo `checkForDeadcode()` e non `checkForDeadcode$0()` poiché i rami `then` ed `else` potrebbero essere comandi composti e non soltanto singoli comandi.

Un ultimo esempio è quello del ciclo `for`. Modifichiamo la classe `absyn/For.java` in accordo con l'Equazione (4.10):

```
public class For extends Command {
  ...
  protected boolean checkForDeadcode$0() {
```

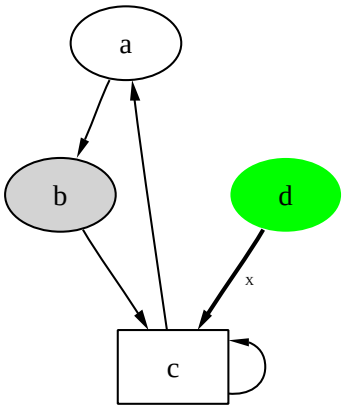


Figura 4.2: Un grafo generato con dot.

```
      update.checkForDeadcode();
      body.checkForDeadcode();
      if (initialisation.checkForDeadcode())
        error("dead-code after for loop initialisation");

      return false;
    }
  }
```

Quello descritto in questa sezione è il controllo di codice morto implementato attualmente dal compilatore Kitten, usato anche per garantire che un metodo non `void` termina sempre con un comando `return`. Compilatori più complessi considerano normalmente altre situazioni in cui è possibile determinare che un comando è codice morto. In particolare, in C o Java è codice morto un comando che segue un `break` o `continue`. La formalizzazione e implementazione di tali controlli *evoluti* è comunque simile a quelle per Kitten.

4.4 Rappresentazione grafica della sintassi astratta

Il linguaggio dot (`http://www.graphviz.org/`) permette di disegnare grafi a partire da una loro specifica testuale data in termini di nodi e archi fra nodi. Si consideri per esempio il grafo in Figura 4.2. Esso è stato generato a partire dal seguente file sorgente `example.dot`, scritto nel linguaggio dot:

```
digraph example {
  size = "11,7.5"

  a [label = "a"]
```

```
  b [label = "b" style = filled]
  c [label = "c" shape = box]
  d [label = "d" color = green style = filled]

  a -> b
  b -> c
  c -> c
  c -> a
  d -> c [label = "X" fontsize = 6 style = bold]
}
```

In tale sorgente si specificano, in un qualsiasi ordine, i nodi e gli archi del grafo. Nodi e archi possono avere proprietà espresse fra parentesi quadre per specificarne la forma, lo spessore o il colore. La trasformazione di tale file sorgente nel file postscript in Figura 4.2 avviene tramite il comando:

```
dot -Tps example.dot >example.ps
```

Ulteriori esempi sono presenti nella pagina web di dot. È possibile avere ulteriori informazioni su dot anche tramite il manuale in linea: `man dot`.

Adesso che abbiamo capito come si specifica un grafo tramite il linguaggio dot, poniamoci il problema di generare un file sorgente dot a partire da una classe di sintassi astratta che descrive una parte di codice Kitten. Per esempio, la sintassi concreta `!this.state` in Figura 1.4 è tradotta nel sottoalbero di sintassi astratta radicato in `Not` in Figura 3.1. Tale sottoalbero è specificato da un file sorgente dot del tipo:

```
node19 [label = "Not"]
node18 [label = "FieldAccess"]
node17 [label = "Variable"]
symbol_this [label = "this" fontname = "Times-Italic" shape = box]
node17 -> symbol_this [label = "name" fontsize = 8]
node18 -> node17 [label = "receiver" fontsize = 8]
symbol_state [label = "state" fontname = "Times-Italic" shape = box]
node18 -> symbol_state [label = "name" fontsize = 8]
node19 -> node18 [label = "expression" fontsize = 8]
```

Si noti che l'esatta numerazione dei nodi non è importante, a differenza della loro etichetta che apparirà nel file postscript finale. Quello che importa però è che tale numerazione generi stringhe diverse per nodi di sintassi astratta diversi, anche se poi la loro etichetta esterna, quella visualizzata nel file postscript, può coincidere. A tal fine, basta che ogni nodo di sintassi astratta contribuisca al file dot con un nodo chiamato `node`*n*, dove *n* è l'identificatore unico di tale nodo (si veda il campo `identifier` in Figura 3.23). I simboli sono invece condivisi e mai ripetuti (Figura 3.25). Essi saranno quindi rappresentati da un nodo chiamato `symbol_`*x*, dove *x* è l'identificatore del simbolo.

La generazione del file sorgente dot è fatta con un algoritmo a discesa ricorsiva: un nodo di sintassi astratta genera una parte del file che descrive se stesso e gli archi verso i suoi figli; quindi chiama ricorsivamente la generazione della parte di file per i suoi figli. Per esempio, il file precedente viene generato in questo modo: il nodo `Not` genera le righe

```
node19 [label = "Not"];
node19 -> node18 [label = "expression" fontsize = 8]
```

Quindi il nodo figlio di `Not`, cioè un `FieldAccess`, genera le righe

```
node18 [label = "FieldAccess"];
node18 -> node17 [label = "receiver" fontsize = 8]
node18 -> symbol_state [label = "name" fontsize = 8]
```

I due nodi figli di `FieldAccess` sono una `Variable`, che genera

```
node17 [label = "Variable"];
node17 -> symbol_this [label = "name" fontsize = 8]
```

e un `Symbol`, che genera

```
symbol_state [label = "state" fontname = "Times-Italic" shape = box]
```

Infine, il figlio di `Variable` è un `Symbol` che genera

```
symbol_this [label = "this" fontname = "Times-Italic" shape = box]
```

Sebbene sia possibile dare una definizione formale della generazione del file dot, limitiamoci per semplicità a mostrarne l'implementazione Java. Per prima cosa, aggiungiamo alcuni metodi di utilità alla classe `absyn/Absyn.java`. Questi sono mostrati in Figura 4.3.

- Il metodo `label()` restituisce l'etichetta che deve essere visualizzata dentro il nodo dot che rappresenta un nodo di sintassi astratta. Normalmente, si tratta del nome della classe di sintassi astratta, senza il prefisso che indica il package `absyn` (per questo si usa `getSimpleName()` piuttosto che `getName()`). Si noti che le sottoclassi potrebbero ridefinire questo metodo. Per esempio, le classi per i letterali ridefiniscono questo metodo in modo da specificare anche il valore lessicale del letterale.

- Il metodo `dotNodeName()` restituisce il nome usato nel file dot per fare riferimento a un nodo di sintassi astratta. Come detto, si tratta della stringa `node`*n* dove *n* è l'identificatore unico del nodo di sintassi astratta.

- Il metodo `linkToNode()` scrive dentro un file testo il comando dot che crea un arco fra il nodo `this` di sintassi astratta e un nodo di sintassi astratta il cui nome usato nel file dot è `to`. Occorre anche specificare l'etichetta `name` dell'arco.

- Il metodo `boldLinkToNode()` si comporta come `linkToNode()` ma crea un arco di maggiore spessore. Questo metodo è usato solo per legare un comando o un membro di una classe al suo successore (campi `next`). Si veda per esempio la Figura 3.1.

```
protected String label() {
  return this.getClass().getSimpleName();
}

protected final String dotNodeName() {
  return "node" + identifier;
}

protected final void linkToNode(String name, String to, FileWriter where)
{
  where.write(dotNodeName() + " -> " + to +
              " [label = \"" + name + "\" fontsize = 8]\n");
}

protected final void boldLinkToNode
  (String name, String to, FileWriter where)
{
  where.write(dotNodeName() + " -> " + to +
              " [label = \"" + name + "\" fontsize = 8 style = bold]\n");
}
```

Figura 4.3: Metodi aggiunti alla classe `absyn/Absyn.java` in Figura 3.23 per la generazione della rappresentazione dot della sintassi astratta.

A questo punto possiamo definire un metodo ricorsivo per la generazione del file dot a partire da un nodo di sintassi astratta. Lo chiameremo `toDot()`. La sua intestazione è:

```
public String toDot(FileWriter where)
```

Questo significa che il metodo scrive dentro il file indicato il codice dot che rappresenta la classe di sintassi astratta e i suoi figli (e ricorsivamente i figli dei figli...). Il valore di ritorno è il nome usato nel file dot per rappresentare il nodo di sintassi astratta su cui il metodo è invocato.

Cominciamo dai tipi. Il metodo `toDot()` è così definito in `absyn/TypeExpression.java`:

```
public final String toDot(FileWriter where) {
  where.write(dotNodeName() + " [ label = \"" + label() + "\"];\n");
  toDot$0(where);
  return dotNodeName();
}

protected void toDot$0(FileWriter where) {}
```

Si noti che `toDot()` è definito come `final`. Esso si limita a generare un nodo dot per il nodo di sintassi astratta e a chiamare un metodo ausiliario `protected` che di default non fa nulla. Le ride-

finizioni del metodo `toDot$0()` creano degli archi verso i nodi figli e richiamano ricorsivamente `toDot()`. Per esempio, eccone la ridefinizione dentro `absyn/ArrayTypeExpression.java`:

```
protected void toDot$0(FileWriter where) {
  linkToNode("elementsType",elementsType.toDot(where),where);
}
```

Lo stesso procedimento si usa per le espressioni. La definizione di `toDot()` dentro alla classe `absyn/Expression.java` è identica al caso dei tipi. La ridefinizione di `toDot$()` dentro `absyn/Variable.java` è per esempio

```
protected void toDot$0(FileWriter where) {
  linkToNode("name",name.toDot(where),where);
}
```

e quella dentro `absyn/FieldAccess.java` è:

```
protected void toDot$0(FileWriter where) {
  linkToNode("receiver",receiver.toDot(where),where);
  linkToNode("name",name.toDot(where),where);
}
```

Come altro esempio, la ridefinizione di `toDot$0()` dentro `absyn/BinOp.java` è

```
protected void toDot$0(FileWriter where) {
  linkToNode("left",left.toDot(where),where);
  linkToNode("right",right.toDot(where),where);
}
```

Vediamo infine la definizione di `toDot()` dentro `symbol/Symbol.java` (Figura 3.25):

```
public String toDot(FileWriter where) {
  where.write("symbol_" + name +
              " [label = \"" + name + "\"" +
              " fontname = \"Times-Italic\" shape = box]\n");
  return "symbol_" + name;
}
```

Questa volta il metodo genera un nodo dot di nome `symbol_`*x*, dove *x* è il nome del simbolo. I simboli non hanno mai archi uscenti, per cui costituiscono un caso base della discesa ricorsiva (le foglie dell'albero in Figura 3.1).

La definizione di `toDot()` è simile per i comandi. Per la struttura complessiva di una classe, il metodo `toDot()` si richiama ricorsivamente sui componenti della classe e aggiunge al file dot un prologo che specifica la dimensione della pagina e un epilogo, cioè la parentesi graffa di chiusura!

L'aggiunta di una nuova classe di sintassi astratta al compilatore Kitten comporta la definizione del suo metodo `toDot$0()`. Come visto in questi esempi, si tratta semplicemente di una sequenza di chiamate a `linkToNode()` per ciascuno dei figli della classe di sintassi astratta che è stata aggiunta. In assenza della specifica del metodo `toDot$0()`, il comportamento di default sarà quello dell'omonimo metodo, vuoto, definito in `absyn/TypeExpression.java` per i tipi, `absyn/Expression.java` per le espressioni e `absyn/Command.java` per i comandi. Conseguentemente non sarà visibile, nel file postscript generato da dot, il sottoalbero radicato nei nodi di sintassi astratta per la nuova classe che è stata aggiunta. Un comportamento indesiderato che non finisce di meravigliare gli studenti...

Esercizio 20. Si formalizzi uno schema a discesa ricorsiva che calcola l'insieme dei nomi di campi letti in un comando. Si faccia lo stesso per l'insieme dei nomi delle classi istanziate in un comando.

Esercizio 21. Si definisca con una discesa ricorsiva un giudizio $\vdash^{\mathsf{simp}}$ tale che $exp \vdash^{\mathsf{simp}} exp'$ è vero quando l'espressione Kitten exp' è ottenuta *semplificando* un'altra espressione Kitten exp. La semplificazione da considerare è quella che sostituisce operazioni binarie fra costanti numeriche con il loro risultato. Una costante numerica è un letterale numerico o un'operazione binaria aritmetica fra costanti numeriche. Si schematizzi quindi l'implementazione Java di $\vdash^{\mathsf{simpl}}$. Riportiamo sotto alcuni esempi di coppie exp ed exp':

exp	exp'
`Addition(IntLiteral(4), IntLiteral(5))`	`IntLiteral(9)`
`LessThan(IntLiteral(4), IntLiteral(5))`	`True()`
`Addition(Multiplication(IntLiteral(4), IntLiteral(5)), Variable(x))`	`Addition(IntLiteral(20), Variable(x))`

Capitolo 5

Analisi Semantica

Le analisi lessicale e sintattica dei Capitoli 2 e 3 garantiscono che il programma sorgente soddisfi le regole di sintassi specificate, rispettivamente, da un insieme di token e da una grammatica. L'analisi sintattica ha inoltre costruito un albero di sintassi astratta che fornisce una visione strutturata del file sorgente (Figura 3.1). Questo non significa che tutti i programmi che hanno superato con successo l'analisi sintattica, cioè senza generare alcun errore di sintassi, siano automaticamente dei programmi *corretti*, pronti ad essere tradotti in codice oggetto ed eseguiti. Per esempio, basta prendere il programma della Figura 1.4 e modificare la linea `this.state := true` in `this.state := 3` per ottenere un programma che supera senza alcun problema sia l'analisi lessicale che quella sintattica, ma che non è *corretto*, poiché esso tenta di assegnare un valore intero (3) a un campo che può contenere solo valori di tipo booleano (`state`). Accorgersi di tali errori va ben al di là delle possibilità delle grammatiche libere dal contesto. Serve uno strumento alternativo, ovvero quello della discesa ricorsiva sull'albero di sintassi astratta del codice sorgente, alla ricerca di errori *semantici* nel codice. Questa *analisi semantica* è l'oggetto di questo capitolo.

Più in dettaglio, i compiti di un'analisi semantica sono quelli di

1. costruire una rappresentazione (una struttura dati) che descrive i tipi usati dal programma (tipi primitivi ma anche array e classi);

2. identificare usi di espressioni incompatibili con i loro tipi statici (*errori di tipo*);

3. identificare occorrenze di variabili usate ma non dichiarate;

4. garantire che un metodo non `void` termini sempre con un comando `return` *exp*, indipendentemente dal percorso di esecuzione che viene seguito al suo interno, e che un metodo `void` non contenga comandi di tipo `return` *exp*;

5. garantire che non ci siano parti di codice che non possono mai essere eseguite e che sono quindi irraggiungibili e *inutili* (identificazione *del codice morto*);

6. identificare e annotare il tipo statico delle espressioni che occorrono in un programma (*inferenza dei tipi*);

7. identificare, per ogni accesso a un campo, la classe in cui il campo è definito;

8. identificare per ogni istruzione `new` *Classe*, sulla base del tipo statico dei parametri attuali, il costruttore di *Classe* che deve essere chiamato in tale punto a tempo di esecuzione;

9. identificare per ogni invocazione di metodo, sulla base del tipo statico dei parametri attuali, la dichiarazione del metodo che deve essere chiamato (o una cui ridefinizione deve essere chiamata) in tal punto a tempo di esecuzione.

Potremmo quindi dire che l'analisi semantica si occupa di costruire una rappresentazione dei tipi usati dal programma (punto 1) che viene poi usata per garantire condizioni di correttezza elementari, senza le quali non ha neppure senso compilare il programma in codice oggetto (*verifica del codice*, punti 2–5) e per raccogliere informazione sul programma che si sta compilando, al fine di facilitare la successiva fase di generazione del codice oggetto (*annotazione del codice*, punti 6–9). Va detto che tale divisione è concettualmente utile ma non netta, dal momento che, per esempio, l'identificazione del costruttore chiamato da un'istruzione `new` (punto 8) è sì un'annotazione utile a generare il codice oggetto che effettua la chiamata a tale costruttore, ma è anche una verifica che tale costruttore esista realmente. L'insieme esatto dei compiti affidati all'analisi semantica varia comunque molto da linguaggio a linguaggio. Altre verifiche effettuate da Java *ma non da Kitten* sono per esempio:

10. garantire che i comandi `break` e `continue` occorrano solo dentro un costrutto iterativo o, per il solo `break`, dentro un comando `switch`;

11. garantire che l'uso di una variabile locale trovi la variabile inizializzata, indipendentemente dal percorso di esecuzione che ha portato al punto di utilizzo della variabile[1].

5.1 I tipi Kitten

Il concetto di *tipo* (Sezione 1.8) è al centro dell'analisi semantica (punti 1,2,4,6,8,9 della precedente enumerazione). Va subito notato che per *tipo* non intendiamo qui la sintassi astratta di una *espressione* di tipo, come nella Sezione 3.6.1. In quel contesto avevamo bisogno di un modo per rappresentare la *struttura sintattica* di una parte di codice che rappresentava un tipo Kitten. Si

[1] In Kitten una variabile va inizializzata al momento della sua dichiarazione, per cui questo controllo è inutile.

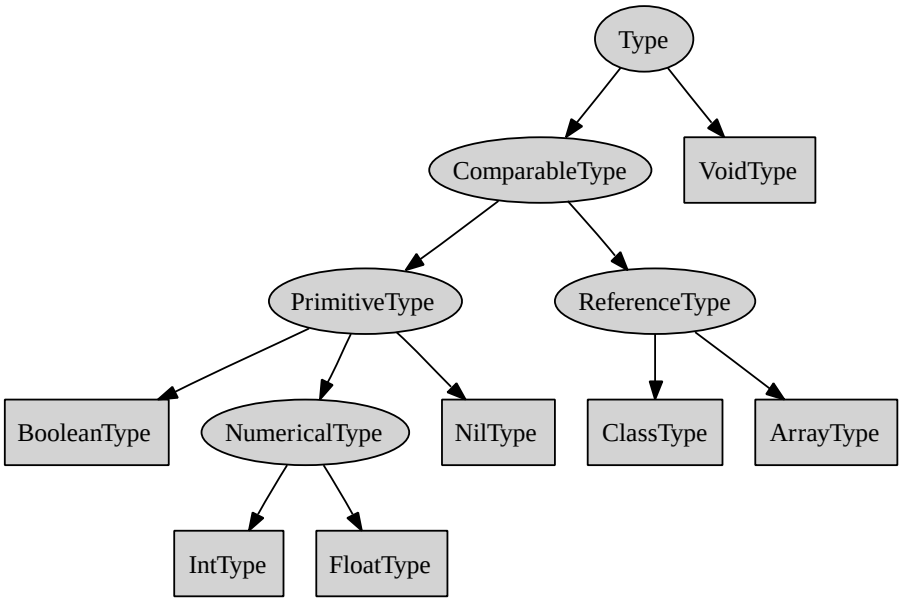


Figura 5.1: Le classi del package `types` che rappresentano i tipi semantici di Kitten.

tratta invece adesso di rappresentare la *struttura semantica* dei tipi delle espressioni Kitten, cioè una struttura dati con associate alcune operazioni che permettono, per esempio, di determinare se un tipo è un sottotipo di un altro o qual è il minimo supertipo comune fra due o più tipi, se esso esiste (Sezione 1.8), o quali sono i campi o i costruttori o metodi di un tipo classe. Per apprezzare la differenza, basta osservare che due occorrenze dell'espressione `int` in due punti diversi di un file sorgente danno origine a due oggetti `IntTypeExpression` diversi, ma il loro tipo semantico è lo stesso identico oggetto.

La distribuzione Kitten contiene il package `types`, al cui interno trovano posto delle classi che rappresentano i tipi *semantici* del linguaggio Kitten. La Figura 5.1 presenta la gerarchia di tali classi. I tipi sono in primo luogo divisi fra *confrontabili* e `void`. I tipi confrontabili sono quelli per i cui valori è definito almeno l'operatore di confronto =. Essi sono a loro volta divisi fra tipi *primitivi* e *riferimento* (Sezione 1.8). I tipi *numerici* sono quei tipi primitivi che rappresentano numeri e per cui sono definite le usuali operazioni di confronto, come il <, oltre a =. Si noti che non esiste un tipo specifico per le stringhe, che sono invece considerate come un caso di `ClassType`.

La Figura 5.2 mostra l'implementazione della superclasse `types/Type.java`. Essa definisce in primo luogo delle costanti per dei tipi di uso comune. Le sue sottoclassi dovranno istanziare il metodo `canBeAssignedTo()` che determina se un tipo può essere assegnato a un altro, seguendo le regole che nella Sezione 1.8 hanno portato alla definizione della relazione $\leq$ sui tipi. Il metodo `canBeAssignedToSpecial()` è per default un sinonimo di `canBeAssignedTo()`, ma viene ridefinito in `types/PrimitiveType.java` e `types/Void.java` come segue:

```
public boolean canBeAssignedToSpecial(Type other) {
```

```
public abstract class Type {
  // delle costanti di uso frequente
  public final static BooleanType BOOLEAN = new BooleanType();
  public final static FloatType FLOAT = new FloatType();
  public final static IntType INT = new IntType();
  public final static NilType NIL = new NilType();
  public final static VoidType VOID = new VoidType();

  protected Type() {}
  public abstract boolean canBeAssignedTo(Type other);
  public boolean canBeAssignedToSpecial(Type other) {
    return canBeAssignedTo(other); // i tipi primitivi lo ridefiniscono
  }
  public Type leastCommonSupertype(Type other) {
    // questo e' ok per i tipi primitivi. Classi e array lo ridefiniscono
    if (this.canBeAssignedTo(other)) return other;
    else if (other.canBeAssignedTo(this)) return this;
    else return null; // non esiste
  }
  public static final ClassType getObjectType() { ... ritorna il tipo per Object }
}
```

Figura 5.2: La superclasse astratta dei tipi semantici di Kitten

```
    return this == other;
  }
```

in modo che i tipi primitivi e `void` siano *sottotipo speciale* solo di se stessi. Questo metodo è utile all'interno della classe `ArrayType`, che vedremo fra un attimo, per implementare la relazione di sottotipaggio $\leq$ che come sappiamo non è monotona sugli array di tipi primitivi (Sezione 1.8). Esso è usato anche per determinare se il tipo di ritorno di una ridefinizione di un metodo è compatibile con quello del metodo ridefinito. Il metodo `leastCommonSupertype()` determina il minimo supertipo comune fra due tipi. Tale supertipo potrebbe non esistere: fra `int` e `boolean` non c'è alcun supertipo comune. La definizione fornita dentro `types/Type.java` funziona per tutti i tipi primitivi, ma come vedremo deve essere ridefinita per i tipi riferimento. Si noti che il costruttore di `types/Type.java` è `protected`. Anche i costruttori delle altre classi che implementano i tipi semantici sono `protected` o `private`. Quindi l'unico modo per ottenere degli oggetti della gerarchia in Figura 5.1 sarà tramite le costanti definite in Figura 5.2 o tramite dei costruttori con memoria che definiremo dentro le classi per i tipi riferimento. Questo implica che *esiste al più un oggetto per un dato tipo semantico* e l'uguaglianza fra tipi può essere controllata con semplici confronti Java ==. Il metodo `getObjectType()` ritorna il tipo della superclasse `Object` di tutte le classi ed array.

Si consideri la classe `types/IntType.java` in Figura 5.3. Come si vede, ammettiamo che il tipo `int` sia assegnato a `int` stesso ma anche a `float`, poiché `int` $\leq$ `float` (Sezione 1.8).

```
public class IntType extends NumericalType {
  protected IntType() {}
  public boolean canBeAssignedTo(Type other) {
    return other == Type.INT || other == Type.FLOAT;
  }
}

public class VoidType extends Type {
  protected VoiType() {}
  public boolean canBeAssignedTo(Type other) { return false; }
  public boolean canBeAssignedToSpecial(Type other) {
      return this == other;
  }
}
```

Figura 5.3: Le classi `types/IntType.java` e `types/VoidType.java` che implementano rispettivamente i tipi `int` e `void`.

La classe `types/VoidType.java` è simile, ma non ammettiamo l'assegnamento verso nessun tipo, neppure `void`. L'assegnamento speciale è invece possibile ma solo verso `void` stesso, come abbiamo già detto.

La scelta di imporre l'uguaglianza nella relazione di sottotipo speciale per i tipi primitivi ha l'effetto che, nel controllo di compatibilità del tipo di ritorno della ridefinizione di un metodo, un tipo primitivo può essere solo sottotipo di se stesso. Si osservi che se `float m()` potesse essere ridefinito, in una sottoclasse, in `int m()` allora una chiamata virtuale del tipo `float f = o.m()` richiederebbe, o meno, una conversione di tipo da `int` a `float` sulla base della classe, a tempo di esecuzione, dell'oggetto contenuto in `o`, il che complica la generazione del codice. Quindi impediamo al programmatore di fare una simile ridefinizione del tipo di ritorno del metodo `m()`. Questo stesso vincolo è imposto nel linguaggio Java.

La classe `types/ArrayType.java` in Figura 5.4 implementa i tipi array. L'invariante che non esistano istanze diverse dello stesso tipo è mantenuta rendendone `private` il costruttore e permettendo la creazione di tipi array solo tramite il metodo statico `mk()`, che usa una memoria per evitare di creare duplicati. L'assegnamento di un tipo array `this` a un altro tipo `other` è considerata legale solo se `other` è `Object` oppure se anche `other` è un tipo array e gli elementi di `this` possono a loro volta essere assegnati a quelli di `other`. Ma si noti l'uso di `canBeAssignedToSpecial()` per questa chiamata ricorsiva! Questo al fine di imporre il vincolo della Sezione 1.8 che richiede che se gli elementi di `this` sono un tipo primitivo allora quelli di `other` devono essere *lo stesso* tipo primitivo.

```
public class ArrayType extends ReferenceType {
  private Type elementsType;
  private ArrayType(Type elementsType) { this.elementsType = elementsType; }
  public static ArrayType mk(Type elementsType) {
    ... usa una memoria per non ricreare tipi array gia' creati in passato
  }
  public boolean canBeAssignedTo(Type other) {
    if (other instanceof ArrayType)
      return elementsType.canBeAssignedToSpecial(((ArrayType)other).elementsType);
    else return other == getObjectType();
  }
  public Type leastCommonSupertype(Type other) {
    // l'lcs fra un array e una classe e' Object
    if (other instanceof ClassType) return getObjectType();
    else if (other instanceof ArrayType)
      if (elementsType instanceof PrimitiveType)
        // fra un array di tipi primitivi e se stesso l'lcs e' l'array.
        if (this == other) return this;
        // fra due array di tipi primitivi diversi, l'lcs e' Object
        else return getObjectType();
      else {
        Type lcs = elementsType.leastCommonSupertype(((ArrayType)other).elementsType);
        if (lcs == null) return getObjectType();
        else return mk(lcs);
      }
    else if (other == Type.NIL) return this; // fra un array e nil e' l'array
    else return null; // non esiste alcun lcs
  }
}
```

Figura 5.4: La classe `types/ArrayType.java` che rappresenta i tipi array.

Questo vincolo, apparentemente strano, è giustificato dal fatto che se `arr` è un array di interi allora il comando `int[] copy := arr` rende `arr` e `copy` *alias*, cioè riferimenti diversi allo stesso oggetto array. Mentre il comando `float[] copy := arr` ci impone di convertire ciascun elemento di `arr` da `int` a `float`. Dal momento che dobbiamo lasciare immutato l'array `arr`, la conversione è possibile solo a costo di creare un *nuovo* array di `float` che contiene i valori convertiti. Tale array verrebbe poi assegnato a `copy`. Ma questo significa che `arr` e `copy` non sarebbero più alias! Detto altrimenti, la scelta del tipo degli elementi di `copy` determinerebbe la condivisione (o meno) fra `arr` e `copy`. Tale comportamento, nettamente inaspettato dal programmatore, è da considerarsi semanticamente pericoloso ed è quindi conveniente vietare tali assegnamenti. Va notato inoltre che il costo computazionale dell'assegnamento diventerebbe lineare nella lunghezza dell'array piuttosto che costante, come normalmente si richiede.

Il metodo `leastCommonSupertype()` di `ArrayType` deve determinare il minimo supertipo comune (*lcs*) fra il tipo array `this` e un altro tipo `other`. Le regole che portano alla definizione di *lcs* sono le seguenti:

- se `other` è una classe allora *lcs* è `Object`. Si noti infatti che tutti gli array e tutte le classi sono sottotipi di `Object` (Sezione 1.8);
- se anche `other` è un tipo array allora:
  - se entrambi sono array dello stesso tipo primitivo allora *lcs* è uguale a `this` (o equivalentemente a `other`);
  - se entrambi sono array di tipi primitivi diversi allora *lcs* è `Object`; si noti che sarebbe errato definire in questo caso *lcs* come `array of Object`, poiché i tipi primitivi non sono sottotipi di `Object`;
  - se entrambi sono array di tipi non primitivi allora *lcs* è il tipo array del minimo supertipo comune fra i tipi degli elementi di `this` e `other`;
- se `other` è il tipo `NilType`, allora *lcs* è `this` poiché `NilType` è un sottotipo di qualsiasi tipo array (Sezione 1.8);
- altrimenti *lcs* non esiste.

Vediamo infine il tipo `ClassType`, che rappresenta i tipi classe come `Object`, `String` o `Led` in Figura 1.4. La Figura 5.5 ne riporta il codice. Il costruttore è lasciato `private` e la costruzione di tipi classe è possibile solo tramite il metodo statico `mk()` che ne garantisce l'unicità. Il costruttore si occupa di creare un analizzatore lessicale per il file sorgente della classe, interfacciarlo con un analizzatore sintattico ed effettuare il parsing sintattico della classe. La sintassi astratta così costruita è memorizzata nel tipo classe. La costruzione prosegue con la superclasse, di cui il nuovo oggetto diventa una sottoclasse. Se non esistesse nessun file col nome della classe seguito da `.kit` o se tale file contenesse degli errori di sintassi, il metodo `parse()` del parser fallirebbe senza restituire alcuna sintassi astratta. Tale eccezione sarebbe allora intercettata da un gestore di eccezioni (non mostrato in Figura 5.5) che fornisce alla classe una sintassi astratta minimale (superclasse `Object`, nessun campo, né costruttori, né metodi). In questo modo si evita di bloccare la compilazione di un programma soltanto perché una della sue classi contiene un errore: si va avanti con la compilazione finché si può, segnalando quanti più errori possibile al programmatore.

Un tipo classe ha informazione sulla *segnatura* della classe, cioè sui suoi campi, costruttori e metodi. Questa informazione è estratta dalla sua sintassi astratta al momento della costruzione di un tipo classe tramite il metodo `addMembers()` a discesa ricorsiva (Figura 5.5). Si noti che se i campi, costruttori o metodi fanno riferimento alla stessa classe che stiamo creando non entriamo in loop poiché abbiamo avuto cura di registrare il tipo classe che stiamo creando nella memoria di `mk()` prima di chiamare `addMembers()`.

La segnatura della classe può essere consultata in seguito tramite dei metodi di ricerca (*lookup*). La differenza fra i metodi `constructorLookup()` e `constructorsLookup()` è che il

```
public class ClassType extends ReferenceType {
  private Symbol name;                    // il nome di questa classe
  private ClassType superclass;           // la sua superclasse (se esiste)
  private ClassTypeList subclasses;       // le sue sottoclassi (se esistono)
  private Parser parser;                  // il parser usato per questa classe
  private ClassDefinition abstractSyntax; // la sintassi astratta della classe

  private ClassType(Symbol name) {
    ... salva this nella memoria usata da mk()
    this.name = name; parser = new Parser(new Lexer(name));
    (abstractSyntax = (ClassDefinition)parser.parse().value).addMembers(this);
    if (name != Symbol.OBJECT) {
      superclass = mk(abstractSyntax.getSuperclassName());
      superclass.subclasses = new ClassTypeList(this,superclass.subclasses);
    }
  }
  public static ClassType mk(Symbol name) {
    ... restituisci l'eventuale classe di nome name contenuta nella memoria
    ... altrimenti restituisci new ClassType(name)
  }
  public FieldSignature fieldLookup(Symbol name) { ... }
  public ConstructorSignature constructorLookup(TypeList formals) { ... }
  public HashSet constructorsLookup(TypeList formals) { ... }
  public MethodSignature methodLookup(Symbol name, TypeList formals) { ... }
  public HashSet methodsLookup(Symbol name, TypeList formals) { ... }
  public boolean canBeAssignedTo(Type other) {
    return other instanceof ClassType && this.subclass((ClassType)other);
  }
  public boolean subclass(ClassType other) {
    return this == other || (superclass != null && superclass.subclass(other));
  }
  public Type leastCommonSupertype(Type other) {
    if (other instanceof ArrayType) return getObjectType();
    else if (other instanceof ClassType)
      for (ClassType cursor = this; cursor != null; cursor = cursor.superclass)
        if (other.canBeAssignedTo(cursor)) return cursor;
    else if (other == Type.NIL) return this; else return null;
  }
}
```

Figura 5.5: La classe `types/ClassType.java` che implementa il tipo classe di Kitten.

primo cerca il costruttore con parametri formali esattamente identici a quelli indicati, mentre il secondo fornisce l'insieme S di *tutti* i costruttori con parametri formali aventi tipi compatibili con quelli indicati. È garantito il vincolo che nessun costruttore in S ha un altro costruttore in S con parametri formali più specifici dei suoi. Per esempio, nella segnatura della classe:

```
class Ambiguous {
  constructor(int i, float d) {}
  constructor(float d, int i) {}
  /* constructor(int i1, int i2) {} */
}
```

il risultato di `constructorsLookup()` con per parametro una lista di due `IntType` è l'insieme dei due costruttori della classe non commentati. Entrambi sono infatti compatibili con due parametri di tipo `int`. Se si eliminasse il commento intorno al terzo costruttore, la stessa chiamata a `constructorsLookup()` restituirebbe un insieme formato dal solo terzo costruttore,

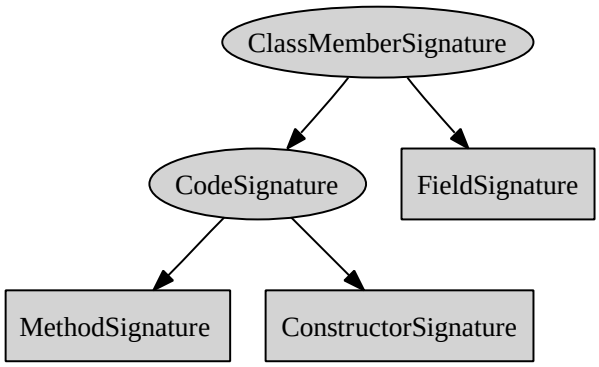


Figura 5.6: Le classi del package `types` che rappresentano le segnature dei membri di una classe.

che è più specifico degli altri due. Si può adesso comprendere a cosa ci servirà il metodo `constructorsLookup()`: di fronte a una invocazione del costruttore di `Ambiguous`, del tipo `new Ambiguous(3,4)`, il compilatore Kitten determina, tramite `constructorsLookup()`, l'insieme dei possibili costruttori candidati a essere chiamati in tale punto di programma. Se ce ne fosse più d'uno, la chiamata verrebbe considerata *ambigua*. Se non ce ne fosse nessuno, la chiamata verrebbe considerata *indefinita*. In entrambi i casi essa verrebbe rifiutata in fase di analisi semantica (lo vedremo quando commenteremo la Figura 5.11). Lo stesso discorso si può fare per `methodLookup()` e `methodsLookup()`, con l'unica differenza che, dal momento che Kitten implementa l'ereditarietà per campi e metodi, la loro ricerca inizia in una data segnatura e, se tale segnatura non definisce esplicitamente il campo o il metodo, la ricerca prosegue ricorsivamente verso l'alto risalendo la catena delle estensioni, verso `Object`.

Il metodo `canBeAssignedTo()` permette l'assegnamento verso la stessa classe o una sua superclasse. Il test di sottoclasse è realizzato dal metodo `subclass()` che scorre verso l'alto a partire da `this` la catena di estensione delle classi alla ricerca dell'ipotetica superclasse. Il metodo `leastCommonSupertype()` determina il minimo supertipo comune *lcs* fra il tipo classe `this` e un altro tipo `other` secondo le regole seguenti:

- se `other` è un tipo array, allora *lcs* è `Object`, poiché tutte le classi e gli array sono sottotipi di `Object` (Sezione 1.8);
- se anche `other` è un tipo classe allora *lcs* è la più specifica superclasse di `this` che è anche superclasse di `other`. Si noti che abbiamo la garanzia che tale *lcs* esista poiché questa ricerca si ferma, nel peggiore dei casi, su `Object`;
- se `other` è il tipo `NilType`, allora *lcs* è `this`, poiché `NilType` è sempre un sottotipo dei tipi classe (Sezione 1.8);
- altrimenti *lcs* non esiste.

La Figura 5.5 mostra che il tipo di ritorno dei metodi di ricerca in una classe è un oggetto (o un insieme di oggetti) di tipo `FieldSignature`, `ConstructorSignature` o `MethodSignature`.

```
public class FieldSignature extends ClassMemberSignature {
  private Type type;   // il tipo del campo
  private Symbol name; // il nome del campo
  public FieldSignature(ClassType clazz, Type type, Symbol name) {
    super(clazz); this.type = type; this.name = name;
  }
}

public abstract class CodeSignature extends ClassMemberSignature {
  private TypeList parameters; // i tipi dei parametri
  protected CodeSignature(ClassType clazz, TypeList parameters) {
    super(clazz); this.parameters = parameters;
  }
}

public class ConstructorSignature extends CodeSignature {
  public ConstructorSignature(ClassType clazz, TypeList parameters) {
    super(clazz,parameters);
  }
}

public class MethodSignature extends CodeSignature {
  private Symbol name;     // il nome del metodo
  private Type returnType; // il suo tipo di ritorno
  public MethodSignature
   (ClassType clazz, Type returnType, TypeList parameters, Symbol name) {
   super(clazz,parameters); this.name = name; this.returnType = returnType;
  }
}
```

Figura 5.7: Le classi che implementano le segnature dei membri di una classe Kitten.

Tali classi implementano le *segnature* di campi, costruttori e metodi, rispettivamente, cioè una specifica delle loro proprietà di tipo. Per esempio, la segnatura di un campo di una classe specifica il nome del campo, il suo tipo semantico di dichiarazione e il tipo semantico della classe in cui il campo è definito.

La Figura 5.6 mostra la gerarchia delle classi del package `types` che rappresentano le segnature di campi, costruttori e metodi Kitten. La Figura 5.7 ne mostra l'implementazione. La superclasse comune `types/ClassMemberSignature.java` (non mostrata in Figura 5.7) descrive la segnatura di un membro di una classe. Essa contiene semplicemente un riferimento al tipo classe a cui il membro appartiene, inizializzato dal costruttore. La classe `FieldSignature` ha in più il tipo e nome del campo descritto. La classe `CodeSignature` ha invece una lista di tipi, corrispondenti ai tipi dei parametri formali del costruttore o metodo che essa rappresenta. La classe `ConstructorSignature` è una estensione di `CodeSignature` che non aggiunge alcun

$$\tau[\![\_]\!] : \texttt{absyn.TypeExpression} \mapsto \texttt{types.Type}$$

$$\begin{aligned}
\tau[\![\texttt{IntTypeExpression()}]\!] &= \texttt{Type.INT} \\
\tau[\![\texttt{FloatTypeExpression()}]\!] &= \texttt{Type.FLOAT} \\
\tau[\![\texttt{BooleanTypeExpression()}]\!] &= \texttt{Type.BOOLEAN} \\
\tau[\![\texttt{VoidTypeExpression()}]\!] &= \texttt{Type.VOID} \\
\tau[\![\texttt{ArrayTypeExpression}(\textit{elementsType})]\!] &= \texttt{ArrayType.mk}(\tau[\![\textit{elementsType}]\!]) \\
\tau[\![\texttt{ClassTypeExpression}(\textit{name})]\!] &= \texttt{ClassType.mk}(\textit{name})
\end{aligned}$$

Figura 5.8: La funzione di analisi semantica $\tau[\![\_]\!]$ per le espressioni di tipo Kitten.

campo, mentre `MethodSignature` specifica anche il nome e il tipo di ritorno del metodo.

5.2 L'analisi semantica delle espressioni di tipo Kitten

Effettuare l'analisi semantica dei tipi Kitten significa costruire il tipo semantico $\tau[\![t]\!]$ rappresentato da ogni espressione sintattica di tipo t che occorre nel programma e annotare tale tipo dentro t stessa. La costruzione di $\tau[\![t]\!]$ è formalizzata nella Figura 5.8. Le espressioni di tipo che rappresentano i tipi primitivi vengono mappate in costanti della classe `types.Type`. Quelle che rappresentano gli array vengono mappate in tipi semantici di tipo `types.ArrayType` per il tipo semantico dei propri elementi. Le espressioni di tipo che rappresentano un tipo classe vengono mappate nell'oggetto di tipo `types.ClassType` per il nome della classe.

La funzione $\tau[\![\_]\!]$ per le espressioni di tipo è implementata tramite un metodo d'istanza di nome `typeCheck()` aggiunto alla classe di sintassi astratta `absyn/TypeExpression.java`:

```
private Type staticType;
public final Type typeCheck() { return staticType = typeCheck$0(); }
protected abstract Type typeCheck$0();
```

Il metodo `typeCheck()`, pubblico e `final`, annota nel campo `staticType` il tipo semantico inferito per l'espressione di tipo. Tale annotazione potrà essere utile in fase di generazione del codice. Lasciamo invece a un metodo `protected` ausiliario `typeCheck$0()` il compito di completare il lavoro con quanto è specifico a ciascuna sottoclasse. Per esempio, per implentare la definizione di $\tau[\![]\!]$ data in Figura 5.8, dentro `absyn/IntTypeExpression.java` ridefiniamo:

```
protected Type typeCheck$0() { return Type.INT; }
```

dentro `absyn/ArrayTypeExpression.java`:

```
protected Type typeCheck$0() {
  return ArrayType.mk(elementsType.typeCheck());
}
```

$$\tau^{\kappa}[\![\_]\!] : \texttt{absyn.ClassMemberDeclaration} \mapsto \texttt{ClassMemberSignature}$$

$$\begin{aligned}
\tau^{\kappa}[\![\texttt{FieldDeclaration}(\mathit{type}, \mathit{name}, \mathit{next})]\!] &= \texttt{new FieldSignature}(\kappa, \tau[\![\mathit{type}]\!], \mathit{name}) \\
\tau^{\kappa}[\![\texttt{ConstructorDeclaration}(\mathit{formals}, \mathit{body}, \mathit{next})]\!] &= \texttt{new ConstructorSignature}(\kappa, \tau[\![\mathit{formals}]\!]) \\
\tau^{\kappa}[\![\texttt{MethodDeclaration}(\mathit{returnType}, \mathit{name}, \mathit{formals}, \mathit{body}, \mathit{next})]\!] & \\
&= \texttt{new MethodSignature}(\kappa, \tau[\![\mathit{returnType}]\!], \tau[\![\mathit{formals}]\!], \mathit{name})
\end{aligned}$$

dove $\tau[\![\texttt{formals}]\!]$ è l'estensione ai parametri formali della funzione $\tau[\![\_]\!]$ della Figura 5.8:

$$\tau[\![\_]\!] : \texttt{absyn.FormalParameters} \mapsto \texttt{types.TypeList}$$

$$\begin{aligned}
\tau[\![\texttt{null}]\!] &= \texttt{null} \\
\tau[\![\texttt{FormalParameters}(\mathit{type}, \mathit{name}, \mathit{next})]\!] &= \texttt{new TypeList}(\tau[\![\mathit{type}]\!], \tau[\![\mathit{next}]\!]) \ .
\end{aligned}$$

Figura 5.9: La funzione $\tau^{\kappa}[\![\_]\!]$ che associa alla sintassi astratta dei membri di una classe κ la loro segnatura.

e dentro `absyn/ClassTypeExpression.java`:

```
protected Type typeCheck$0() { return ClassType.mk(name); }
```

Possiamo adesso mostrare in Figura 5.9 una funzione $\tau^{\kappa}[\![\_]\!]$ che costruisce le segnature dei membri di una classe κ a partire dalla loro sintassi astratta. Questa funzione è implementata dal metodo `addMembers()` usato al momento della costruzione di un tipo classe per arricchirlo con le segnature dei suoi membri (Figura 5.5).

5.3 L'analisi semantica delle espressioni Kitten

L'analisi semantica delle espressioni Kitten consiste nell'annotare a tempo di compilazione ciascuna espressione e che occorre nel programma sorgente con il suo tipo statico t_e (Sezione 1.8). Essendo Kitten un linguaggio fortemente tipato, occorre definire t_e in modo che, a tempo di esecuzione, il tipo dinamico di e (cioè il tipo del valore di e, Sezione 1.8) sia t_e o un sottotipo di t_e. L'analisi semantica deve inoltre garantire che i tipi siano usati correttamente dentro e. Deve rifiutare per esempio espressioni del tipo `3+l` dove `l` è una variabile dichiarata di tipo `Led` (Figura 1.4). Deve anche determinare il costruttore o metodo che deve essere chiamato a tempo di esecuzione dalle espressioni `new` o dalle invocazioni di metodo contenute in e. Per esempio, deve determinare che l'espressione `l.isOn()` chiama il metodo `isOn()` della Figura 1.4 o una delle ridefinizioni di tale metodo nelle sottoclassi di `Led` (se mai ne esistessero). Questo è essenziale sia per determinare il tipo statico dell'espressione `l.isOn()` (che sarà il tipo di ritorno

$$\frac{\rho(\mathit{name}) \text{ è definito}}{\rho \vdash \texttt{Variable}(\mathit{name}) : \rho(\mathit{name})} \qquad \frac{\rho \vdash \mathit{receiver} : \kappa \quad \kappa \in \texttt{ClassType} \quad \mathit{field} = \kappa.\texttt{fieldLookup}(\mathit{name}) \quad \mathit{field} \neq \texttt{null}}{\rho \vdash \texttt{FieldAccess}(\mathit{receiver}, \mathit{name}) : \mathit{field}.\texttt{getType}()}$$

$$\frac{\rho \vdash \mathit{array} : t \quad t \in \texttt{ArrayType} \quad \rho \vdash \mathit{index} : \texttt{Type.INT}}{\rho \vdash \texttt{ArrayAccess}(\mathit{array}, \mathit{index}) : t.\texttt{getElementsType}()}$$

$$\frac{}{\rho \vdash \texttt{False}() : \texttt{Type.BOOLEAN}} \qquad \frac{}{\rho \vdash \texttt{True}() : \texttt{Type.BOOLEAN}} \qquad \frac{}{\rho \vdash \texttt{Nil}() : \texttt{Type.NIL}}$$

$$\frac{}{\rho \vdash \texttt{IntLiteral}() : \texttt{Type.INT}} \qquad \frac{}{\rho \vdash \texttt{FloatLiteral}() : \texttt{Type.FLOAT}}$$

$$\frac{}{\rho \vdash \texttt{StringLiteral}(\mathit{value}) : \texttt{ClassType.mk}(\texttt{Symbol.STRING})}$$

Figura 5.10: Le regole per l'analisi semantica dei leftvalue e dei letterali Kitten.

di `isOn()` in Figura 1.4, cioè `boolean`) che per garantire, a tempo di compilazione, che tale chiamata di metodo non terminerà mai, a tempo di esecuzione, con un'eccezione causata dalla mancata identificazione di un metodo da invocare (questa garanzia è possibile per Kitten poiché esso non ammette il caricamento dinamico delle classi. Non è invece possibile per Java che lo ammette). Infine, tale controllo è utile in vista della generazione del codice intermedio (Capitolo 6), momento in cui sapremo già con quale codice (o insiemi di codici, nel caso di chiamate virtuali) legare questa invocazione di metodo. Un discorso analogo si può fare per gli accessi ai campi delle classi, per i quali l'analisi semantica deve identificare la classe che definisce il campo a cui si fa accesso.

Effettueremo il controllo semantico di un'espressione e tramite un giudizio $\vdash e : t_e$ definito a discesa ricorsiva sulla sintassi astratta delle espressioni. Gli esempi precedenti mostrano però che a tal fine avremo bisogno di conoscere il tipo di dichiarazione delle variabili in scope nel punto di programma in cui e occorre, al fine di determinare il tipo delle variabili contenute in e. Estendiamo quindi il nostro giudizio in $\rho \vdash e : t_e$, dove ρ è un *ambiente* o *contesto*. Formalmente $\rho : V \mapsto \texttt{types.Type}$, dove V è l'insieme delle variabili che sono in scope nel punto di programma in cui e occorre. La definizione di questo giudizio è in Figura 5.10 per quanto riguarda i leftvalue e i letterali Kitten e in Figura 5.11 per le restanti espressioni. Si noti subito che il giudizio $\rho \vdash e : t_e$ non è sempre definito. Si deve immaginare che, quando esso non è definito, un messaggio di errore viene comunicato al programmatore.

Consideriamo adesso le regole più significative delle Figure 5.10 e 5.11.

`Variable`(*name*)**.** Abbiamo già osservato che l'ambiente ρ serve proprio a specificare il tipo di dichiarazione delle variabili in scope nel punto di programma in cui occorre l'espressione che stiamo analizzando. In questo caso, quindi, basta leggere il tipo di dichiarazione di *name* per determinare il tipo statico di `Variable`(*name*). Questo è in effetti l'unico caso in cui usiamo esplicitamente l'ambiente ρ. Negli altri casi ci limiteremo a passarlo ricorsivamente alle componenti dell'espressione che stiamo analizzando.

`FieldAccess`(*receiver*, *name*)**.** L'accesso al campo di nome *name* dell'oggetto contenuto nel-

$$\frac{\texttt{ClassType.mk}(\textit{className}) = \kappa \quad \rho \vdash \textit{actuals} : \vec{\tau} \quad \kappa.\texttt{constructorsLookup}(\vec{\tau}) = \{\textit{constructor}\}}{\rho \vdash \texttt{NewObject}(\textit{className}, \textit{actuals}) : \kappa}$$

$$\frac{\rho \vdash \textit{elementsType} : t \quad \rho \vdash \textit{size} : \texttt{Type.INT}}{\rho \vdash \texttt{NewArray}(\textit{elementsType}, \textit{size}) : \texttt{ArrayType.mk}(t)}$$

$$\frac{\begin{array}{c}\rho \vdash \textit{receiver} : \kappa \quad \kappa \in \texttt{ClassType} \quad \rho \vdash \textit{actuals} : \vec{\tau} \\ \kappa.\texttt{methodsLookup}(\textit{name}, \vec{\tau}) = \{\textit{method}\} \quad r = \textit{method}.\texttt{getReturnType}() \quad r \neq \texttt{Type.VOID}\end{array}}{\rho \vdash \texttt{MethodCallExpression}(\textit{receiver}, \textit{name}, \textit{actuals}) : r}$$

$$\frac{\rho \vdash \textit{expression} : \texttt{Type.BOOLEAN}}{\rho \vdash \texttt{Not}(\textit{expression}) : \texttt{Type.BOOLEAN}} \qquad \frac{\rho \vdash \textit{expression} : t \quad t \leq \texttt{Type.FLOAT}}{\rho \vdash \texttt{Minus}(\textit{expression}) : t}$$

$$\frac{\textit{intoType} = \tau[\![\textit{type}]\!] \quad \rho \vdash \textit{expression} : \textit{fromType} \quad \textit{intoType} < \textit{fromType}}{\rho \vdash \texttt{Cast}(\textit{type}, \textit{expression}) : \textit{intoType}}$$

$$\frac{\rho \vdash \textit{left} : \texttt{Type.BOOLEAN} \quad \rho \vdash \textit{right} : \texttt{Type.BOOLEAN}}{\rho \vdash \texttt{BooleanBinOp}(\textit{left}, \textit{right}) : \texttt{Type.BOOLEAN}}$$

$$\frac{\rho \vdash \textit{left} : t_l \quad \rho \vdash \textit{right} : t_r \quad t_l \leq \texttt{Type.FLOAT} \quad t_r \leq \texttt{Type.FLOAT}}{\rho \vdash \texttt{ArithmeticBinOp}(\textit{left}, \textit{right}) : t_l.\texttt{leastCommonSupertype}(t_r)}$$

$$\frac{\rho \vdash \textit{left} : t_l \quad \rho \vdash \textit{right} : t_r \quad t_l \leq \texttt{Type.FLOAT} \quad t_r \leq \texttt{Type.FLOAT}}{\rho \vdash \texttt{NumericalComparisonBinOp}(\textit{left}, \textit{right}) : \texttt{Type.BOOLEAN}}$$

$$\frac{\rho \vdash \textit{left} : t_l \quad \rho \vdash \textit{right} : t_r \quad (t_l \leq t_r \text{ oppure } t_r \leq t_l)}{\rho \vdash \texttt{Equal}(\textit{left}, \textit{right}) : \texttt{Type.BOOLEAN}}$$

$$\frac{\rho \vdash \textit{left} : t_l \quad \rho \vdash \textit{right} : t_r \quad (t_l \leq t_r \text{ oppure } t_r \leq t_l)}{\rho \vdash \texttt{NotEqual}(\textit{left}, \textit{right}) : \texttt{Type.BOOLEAN}}$$

Figura 5.11: Le regole per l'analisi semantica delle restanti espressioni Kitten.

l'espressione *receiver* richiede in primo luogo di determinare il tipo statico κ di *receiver*. La precondizione richiede che κ sia un tipo classe, poiché in Kitten solo le classi hanno campi. L'ulteriore richiesta è che κ abbia effettivamente un campo di nome *name*, definito da κ stesso o ereditato da una superclasse di κ. Questo si può verificare con il metodo `fieldLookup()` a partire dalla segnatura di κ (Figura 5.5). Il risultato di tale metodo è la segnatura *field* del campo a cui si sta facendo riferimento. Il tipo dell'espressione di accesso al campo è quindi il tipo di dichiarazione del campo, ottenibile come *field*`.getType()`.

`ArrayAccess`(*array*, *index*)**.** L'accesso a un elemento di un array richiede di effettuare ricorsivamente l'analisi semantica dell'espressione *array* che contiene l'array a cui si accede e dell'espressione *index* che contiene l'indice in cui si accede nell'array. Si richiede come precondizione che *array* abbia tipo array t e che *index* abbia tipo `int`. Il tipo dell'accesso all'array è il tipo degli elementi di t, cioè t`.getElementsType()`.

`NewObject`(*className*, *actuals*)**.** Il tipo statico di questa espressione, che crea un oggetto di classe *className*, è il tipo classe κ di nome *className*: κ = `ClassType.mk`(*className*). Occorre però controllare che non ci siano errori semantici nei parametri attuali *actuals*. Questo si ottiene richiamando ricorsivamente su di essi l'analisi semantica, cioè verificando il giudizio $\rho \vdash actuals : \vec{\tau}$, che è l'estensione del giudizio $\rho \vdash e : t_e$ a sequenze di espressioni:

$$\rho \vdash \texttt{null} : \texttt{null}$$

$$\frac{\rho \vdash head : h \quad \rho \vdash tail : \vec{\tau}}{\rho \vdash \texttt{ExpressionSeq}(head, tail) : \texttt{new TypeList}(h,\vec{\tau})}$$

Occorre anche garantire che fra i costruttori di κ che possono essere chiamati con parametri attuali di tipo $\vec{\tau}$ ce ne sia uno più specifico degli altri. Questo si verifica chiamando il metodo `constructorsLookup`($\vec{\tau}$) sulla classe κ (Figura 5.5) e controllando che il risultato sia un insieme di un solo elemento.

`MethodCallExpression`(*receiver*, *name*, *actuals*)**.** L'analisi semantica dell'invocazione di un metodo richiede in primo luogo di effettuare ricorsivamente l'analisi semantica del ricevitore e dei parametri attuali dell'invocazione, cioè di verificare i giudizi $\rho \vdash receiver : \kappa$ e $\rho \vdash actuals : \vec{\tau}$ (quest'ultimo è l'estensione di $\rho \vdash e : t_e$ a sequenze di espressioni, si veda sopra il caso di `NewObject`). Si richiede che κ sia un tipo classe, poiché solo le classi hanno metodi in Kitten. Inoltre fra i metodi definiti o ereditati da κ e che possono essere chiamati con parametri attuali di tipo statico $\vec{\tau}$ ne deve esistere uno che è più specifico di tutti gli altri. Questo si ottiene chiamando il metodo `methodsLookup`($\vec{\tau}$) sulla classe κ (Figura 5.5) e verificando che il risultato sia un insieme di un solo elemento, la `MethodSignature` *method*. Il tipo statico dell'invocazione di metodo è quindi il tipo del valore ritornato dal metodo, cioè *method*.`getReturnType`(). Si richiede che tale tipo non sia `void` poiché un'espressione deve avere un valore associato a tempo di esecuzione.

`Cast`(*type*, *expression*)**.** Quest'espressione effettua il cast di *expression* verso il tipo *type*. La sua analisi semantica effettua ricorsivamente l'analisi semantica di *type* ed *expression* e poi richiede che il tipo semantico di *type* sia un sottotipo stretto del tipo semantico di *expression*. Questo vincolo accetta quindi esclusivamente cast *verso il basso* scartando per esempio espressioni come `3 as Persona`, `3 as float`, `3 as int` o `studente as Persona`. Il motivo per cui tali cast sono rifiutati è che sarebbero impossibili (come nell'esempio `3 as Persona`) oppure sempre veri: è sempre possibile usare un intero dove serve un valore in virgola mobile o un intero; è sempre possibile usare uno studente dove serve una persona. Rifiutando questi ultimi cast si obbliga il programmatore a scrivere del codice più semplificato (`3` al posto di `3 as float` e di `3 as int`, `studente` al posto di `studente as Persona`).

`ArithmeticBinOp`(*left*, *right*)**.** L'analisi semantica di un'operazione binaria aritmetica effettua ricorsivamente l'analisi semantica dei suoi due operandi, cioè verifica i giudizi $\rho \vdash left : t_l$ e $\rho \vdash right : t_r$. Tali due espressioni devono avere un tipo statico che sia `int` o `float`. Il tipo

statico del risultato dell'operazione è il minimo supertipo comune fra t_l e t_r. Questo significa per esempio che `3 + 4` ha tipo statico `int` e `3 + 4.5` ha tipo statico `float`. Si noti che dando una regola per la classe astratta delle operazioni binarie aritmetiche non abbiamo bisogno di specificare esplicitamente alcuna regola per le sue sottoclassi (Figura 3.26).

`NumericalComparisonBinOp`(*left*, *right*)**.** Il ragionamente è simile a quello per le espressioni aritmetiche binarie, con l'unica differenza che il risultato di un confronto fra due espressioni è sempre un booleano.

`Equal`(*left*, *right*) **e** `NotEqual`(*left*, *right*)**.** L'analisi semantica dell'uguaglianza e della disuguaglianza fra due espressioni richiede di effettuarne ricorsivamente l'analisi semantica e impone che il tipo di una delle due espressioni sia un sottotipo (non stretto) di quello dell'altra. Questo vincolo serve a rifiutare espressioni di uguaglianza che non potrebbero mai essere vere ed espressioni di disuguaglianza che sarebbero sempre false. Per esempio, se `p` è una variabile di classe `Persona`, sottoclasse diretta di `Object`, e `c` è una variabile di classe `Automobile`, anch'essa sottoclasse diretta di `Object`, allora l'uguaglianza `p = c` è sempre falsa, poiché non esisterà mai un oggetto che sia al contempo una `Persona` e un'`Automobile`. Per lo stesso motivo, la disuguaglianza `p != c` è sempre vera. Rifiutando queste espressioni costringiamo il programmatore a eliminare dal suo programma dei test inutili.

5.3.1 L'implementazione dell'analisi semantica delle espressioni

L'implementazione delle regole nelle Figure 5.10 e 5.11 richiede in primo luogo di implementare l'ambiente ρ. Si potrebbe pensare di utilizzare una semplice `java.util.HashMap` che lega le variabili ai loro tipi di dichiarazione. Ma fra poco (Sezione 5.4) avremo bisogno di un'operazione di estensione *non distruttiva* sugli ambienti, tale cioè da lasciare il vecchio ambiente intatto dopo la sua estensione. Questo rende l'uso di `java.util.HashMap` sconveniente, poiché tale struttura dati ha solo operazioni distruttive. Decidiamo quindi di usare una nostra struttura dati per rappresentare gli ambienti, cioè la classe `symbol/Table.java` e le sue due sottoclassi in Figura 5.12. L'interfaccia `symbol.Table` specifica semplicemente che un ambiente ha un'operazione `get(key)` che permette di leggere il valore di una variabile `key` e un'operazione `put(key,value)` che costruisce un nuovo ambiente in cui la variabile `key` è legata a `value`. Si noti che le variabili sono genericamente legate a degli `Object`, benché a noi servirebbero degli ambienti che legano le variabili a dei `types.Type`. Questo dà maggiore generalità a questi ambienti, che in futuro potrebbero essere usati per altri scopi, in cui alle variabili sono legate strutture dati diverse da `types.Type`. Si noti inoltre che il metodo `put()` restituisce un *nuovo* ambiente con aggiunto un nuovo legame: il vecchio ambiente non è modificato ed è ancora utilizzabile. Questo al fine di implementare un'operazione `put()` non distruttiva, come volevamo.

Le sottoclassi di `symbol.Table` sono `symbol.EmptyTable` e `symbol.NonEmptyTable`. La prima implementa un ambiente vuoto in cui non esiste alcun legame per le variabili. La seconda implementa un ambiente in cui c'è almeno un legame per una variabile. Questo ambiente è rappresentato come un albero binario di ricerca, in cui cioè le variabili che precedono la radice,

```
public abstract class Table {
  public final static EmptyTable EMPTY = new EmptyTable();
  public abstract Object get(Symbol key);
  public abstract Table put(Symbol key, Object value);
}

class EmptyTable extends Table {
  public Object get(Symbol key) { return null; }
  public Table put(Symbol key, Object value) {
    return new NonEmptyTable(key,value);
  }
}

class NonEmptyTable extends Table {
  private Symbol key;  private Object value;  private Table left, right;

  private NonEmptyTable(Symbol key, Object value, Table left, Table right) {
    this.key = key; this.value = value;
    this.left = left; this.right = right;
  }
  NonEmptyTable(Symbol key, Object value) { this(key,value,EMPTY,EMPTY); }
  public Object get(Symbol key) {
    int comp = this.key.compareTo(key);
    if (comp < 0) return left.get(key);
    else if (comp == 0) return value;
    else return right.get(key);
  }
  public Table put(Symbol key, Object value) {
    Table temp; int comp = this.key.compareTo(key);
    if (comp < 0) {
      temp = left.put(key,value);
      if (temp == left) return this;
      else return new NonEmptyTable(this.key,this.value,temp,right);
    }
    else if (comp == 0)
      if (value == this.value) return this;
      else return new NonEmptyTable(this.key,this.value,left,right);
    else {
      temp = right.put(key,value);
      if (temp == right) return this;
      else return new NonEmptyTable(this.key,this.value,left,temp);
    }
  }
}
```

Figura 5.12: Le classi del package `symbol` che implementano gli ambienti.

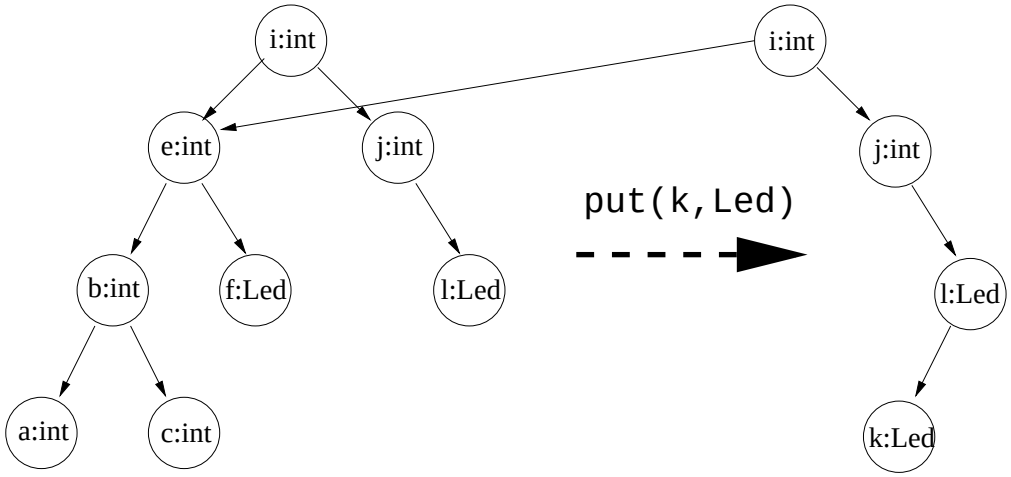


Figura 5.13: L'inserzione non distruttiva in un ambiente di un legame per una variabile.

in ordine lessicografico, vanno cercate nel sottoalbero di sinistra e quelle che la seguono vanno cercate nel sottoalbero destro. Questo è proprio quello che fa il metodo `get()` (Figura 5.12). Il metodo `put()` invece costruisce un *nuovo* albero binario in cui la variabile è legata al valore passato come argomento, senza modificare l'albero orginale. Esso implementa quindi un'inserzione non distruttiva. La Figura 5.13 mostra come è effettuata l'inserzione. Al posto di ricreare integralmente l'albero binario, se ne condivide una gran parte, ricostruendo solo il cammino dalla radice dell'albero al nodo che è stato aggiunto o modificato.

Gli ambienti sono contenuti dentro un *type-checker*, il quale è implementato dalla classe `semantical/TypeChecker.java` in Figura 5.14. Per adesso l'ambiente è tutto quello di cui abbiamo bisogno per effettuare l'analisi semantica delle espressioni, ma per i comandi aggiungeremo al type-checker ulteriori informazioni (Sezione 5.4).

Possiamo a questo punto implementare le regole delle Figure 5.10 e 5.11 tramite una discesa ricorsiva sulla sintassi astratta delle espressioni. In `absyn/Expression.java` aggiungiamo:

```
private Type staticType;
private TypeChecker checker;

public final Type typeCheck(TypeChecker checker) {
  return staticType = typeCheck$0(this.checker = checker);
}

protected abstract Type typeCheck$0(TypeChecker checker);
```

Il metodo `public` e `final`, di nome `typeCheck()`, effettua le operazioni comuni a tutte le espressioni, cioè l'annotazione del tipo statico inferito per l'espressione e del type-checker usato

```
public class TypeChecker {
  private Table env;
  private ErrorMsg errorMsg;

  private TypeChecker(Table env, ErrorMsg errorMsg) {
    this.env = env; this.errorMsg = errorMsg;
  }
  public TypeChecker(ErrorMsg errorMsg) {
    this(Table.EMPTY,errorMsg);
  }
  public TypeChecker putVar(Symbol var, Type type) {
    return new TypeChecker(env.put(var,type),errorMsg));
  }
  public Type getVar(Symbol var) { return (Type)env.get(var); }
  public void error(int pos, String msg) { errorMsg.error(pos,msg); }
}
```

Figura 5.14: Il type-checker usato per effettuare l'analisi semantica delle espressioni Kitten.

per inferirlo. Un metodo ausiliario e `protected`, di nome `typeCheck$0()`, implementa le operazioni specifiche alla singola espressione, come specificate nelle Figure 5.10 e 5.11.

Vediamo alcuni esempi di implementazione del metodo `typeCheck$0()`. Dentro la classe `absyn/Variable.java` definiamo:

```
protected Type typeCheck$0(TypeChecker checker) {
  Type result = checker.getVar(name);
  if (result == null) return error("undefined variable " + name);
  else return result;
}
```

Questa implementazione riflette la specifica in Figura 5.10: si cerca la variabile nell'ambiente; se non esiste si dà un errore altrimenti se ne restituisce il tipo. Il metodo `error()` è definito dentro `absyn/Expression.java` come

```
protected Type error(String msg) {
  error(checker,msg);
  return Type.INT;
}
```

Esso stampa il messaggio di errore tramite il type-checker in utilizzo per l'espressione e ritorna il tipo di emergenza `int`. Questo permette di continuare il type-checking anche in presenza di un errore, benché possa causare degli errori di tipo a cascata. Il metodo `error()` a due argomenti è poi definito dentro `absyn/Absyn.java` come

```
protected void error(TypeChecker checker, String msg) {
```

```
    checker.error(pos,msg);
  }
```

Esso usa il campo `pos` della sintassi astratta per indicare in che punto dare l'errore all'utente. Tale campo era il numero di caratteri passati dall'inizio del file a un punto significativo della parte di sintassi astratta in considerazione (Sezione 3.6).

Esaminiamo un altro esempio, quello di `absyn/FieldAccess.java`:

```
protected Type typeCheck$0(TypeChecker checker) {
  Type receiverType = receiver.typeCheck(checker);
  if (!(receiverType instanceof ClassType))
    return error("class type required");
  ClassType receiverClass = (ClassType)receiverType;
  if ((field = receiverClass.fieldLookup(name)) == null)
    return error("unknown field " + name);
  return field.getType();
}
```

Consistentemente con la Figura 5.10, tale metodo effettua ricorsivamente l'analisi semantica di `receiver` e quindi impone che esso abbia un tipo classe. Infine cerca il campo di nome `name` dentro tale tipo classe e ne restituisce il tipo.

Un altro esempio è quello di `absyn/ArrayAccess.java`:

```
protected Type typeCheck$0(TypeChecker checker) {
  Type arrayType = array.typeCheck(checker);
  index.mustBeInt(checker);
  if (!(arrayType instanceof ArrayType))
    return error("array type required");
  return ((ArrayType)arrayType).getElementsType();
}
```

Consistentemente con la Figura 5.10, tale metodo effettua ricorsivamente l'analisi semantica di `array` e `index`. Per `index` usa il metodo ausiliario `mustBeInt()` che è definito dentro `absyn/Expression.java` come:

```
protected void mustBeInt(TypeChecker checker) {
  if (typeCheck(checker) != Type.INT) error("integer expected");
}
```

Consideriamo infine la definizione di `typeCheck$0()` in `absyn/ArithmeticBinOp.java`:

```
protected Type typeCheck$0(TypeChecker checker) {
  Type leftType = getLeft().typeCheck(checker);
  Type rightType = getRight().typeCheck(checker);
  if (leftType.canBeAssignedTo(Type.FLOAT) &&
      rightType.canBeAssignedTo(Type.FLOAT))
```

$$\frac{}{\rho \vdash \texttt{Skip}() : \rho} \qquad \frac{\rho \vdash \mathit{condition} : \texttt{Type.BOOLEAN} \quad \rho \vdash \mathit{then} : \rho' \quad \rho \vdash \mathit{else} : \rho''}{\rho \vdash \texttt{IfThenElse}(\mathit{condition}, \mathit{then}, \mathit{else}) : \rho}$$

$$\frac{\mathit{expression} \neq \texttt{null} \quad \rho \vdash \mathit{expression} : t \quad \text{il comando occorre in un metodo che ritorna } r \quad t \leq r}{\rho \vdash \texttt{Return}(\mathit{expression}) : \rho}$$

$$\frac{\text{il comando occorre in un costruttore o in un metodo che ritorna } \texttt{void}}{\rho \vdash \texttt{Return}(\texttt{null}) : \rho}$$

$$\frac{\rho \vdash \mathit{lvalue} : t_l \quad \rho \vdash \mathit{rvalue} : t_r \quad t_r \leq t_l}{\rho \vdash \texttt{Assignment}(\mathit{lvalue}, \mathit{rvalue}) : \rho}$$

$$\frac{\rho \vdash \mathit{initialisation} : \rho' \quad \rho' \vdash \mathit{condition} : \texttt{Type.BOOLEAN} \quad \rho' \vdash \mathit{update} : \rho'' \quad \rho' \vdash \mathit{body} : \rho'''}{\rho \vdash \texttt{For}(\mathit{initialisation}, \mathit{condition}, \mathit{update}, \mathit{body}) : \rho}$$

$$\frac{\rho \vdash \mathit{condition} : \texttt{Type.BOOLEAN} \quad \rho \vdash \mathit{body} : \rho'}{\rho \vdash \texttt{While}(\mathit{condition}, \mathit{body}) : \rho}$$

$$\frac{t = \tau[\![\mathit{type}]\!] \quad \rho \vdash \mathit{initialiser} : i \quad i \leq t}{\rho \vdash \texttt{LocalDeclaration}(\mathit{type}, \mathit{name}, \mathit{initialiser}) : \rho[\mathit{name} \mapsto t]}$$

$$\frac{\rho \vdash \mathit{body} : \rho'}{\rho \vdash \texttt{LocalScope}(\mathit{body}) : \rho} \qquad \frac{\begin{array}{c}\rho \vdash \mathit{receiver} : \kappa \quad \kappa \in \texttt{ClassType} \quad \rho \vdash \mathit{actuals} : \vec{\tau} \\ \kappa.\texttt{methodsLookup}(\mathit{name}, \vec{\tau}) = \{\mathit{method}\}\end{array}}{\rho \vdash \texttt{MethodCallCommand}(\mathit{receiver}, \mathit{name}, \mathit{actuals}) : \rho}$$

$$\frac{\rho \vdash c_1 : \rho' \quad \rho' \vdash c_2 : \rho''}{\rho \vdash c_1; c_2 : \rho''}$$

Figura 5.15: Le regole per l'analisi semantica dei comandi Kitten.

```
    return leftType.leastCommonSupertype(rightType);
  else return error("numerical argument required");
}
```

Consistentemente con la Figura 5.11, esso effettua ricorsivamente l'analisi semantica di `left` e `right` e impone che abbiano un tipo statico che sia `float` o un sottotipo di `float`. Il tipo statico dell'operazione binaria è il minimo supertipo comune fra i tipi statici di `left` e `right`.

5.4 L'analisi semantica dei comandi Kitten

La Figura 5.15 mostra le regole di analisi semantica per i comandi Kitten. Questa volta usiamo un giudizio $\rho \vdash c : \rho'$ il cui significato è che il comando c eseguito a partire da un ambiente ρ porta in un ambiente ρ'. Questo perché i comandi non hanno un valore ma possono modificare l'ambiente e le uniche modifiche visibili al livello dei tipi sono quelle dell'insieme e del tipo delle variabili in scope. In particolare, è la dichiarazione di una variabile (la `LocalDeclaration` in Figura 5.15) che estende l'ambiente con una nuova variabile locale, che sostituisce eventualmente una variabile già in scope e con lo stesso nome.

Esaminiamo adesso alcune regole della Figura 5.15:

`IfThenElse`(*condition*, *then*, *else*)**.** Il condizionale richiede che la condizione sia un'espressione di tipo booleano ed effettua ricorsivamente l'analisi semantica di *then* ed *else*. La scelta di lasciare ρ immutato come risultato dell'analisi semantica del condizionale implica che eventuali variabili locali dichiarate all'interno del ramo *then* o del ramo *else* non sono più in scope alla fine del condizionale.

`Return`(*expression*)**.** Il comando di ritorno da metodo o costruttore richiede di effettuare ricorsivamente l'analisi semantica dell'espressione ritornata, se esiste. Nel caso in cui essa non sia `null`, allora questo comando deve occorrere dentro un metodo che ritorna il tipo statico di *expression* o un suo supertipo. Altrimenti questo comando deve occorrere dentro un metodo che ritorna `void` o dentro un costruttore.

`Assignment`(*lvalue*, *rvalue*)**.** L'analisi semantica dell'assegnamento del valore di un'espressione a un leftvalue consiste nel controllare che il tipo statico dell'espressione sia lo stesso o un sottotipo del tipo statico del leftvalue.

`For`(*initialisation*, *condition*, *update*, *body*)**.** L'analisi semantica del ciclo `for` comincia analizzando ricorsivamente il comando *initialisation*. Il risultato di questa analisi è un ambiente ρ', eventualmente arricchito, rispetto a ρ, con una dichiarazione di una variabile locale al ciclo. Si impone poi che *condition* abbia tipo booleano. L'ambiente ρ' viene usato per effettuare l'analisi semantica di *initialisation*, *update* e *body*, al fine di permettere al programmatore di dichiarare una variabile locale dentro *initialisation* e di usarla nelle altre componenti del `for`, come in

```
for (int i := 0; i < 5; i := i + 1) "".concat(i).output()
```

Se si fosse usato ρ per l'analisi di *condition*, *update* e *body*, la variabile `i` sarebbe risultata indefinita o avrebbe fatto riferimento a un'altra variabile, definita esternamente al ciclo.

`LocalDeclaration`(*type*, *name*, *initialiser*)**.** L'analisi semantica della dichiarazione di una variabile locale estende l'ambiente legando la variabile *name* al tipo semantico di *type*. Si effettua anche ricorsivamente l'analisi semantica di *initialiser* e si impone che il suo tipo statico sia lo stesso o un sottotipo del tipo semantico di *type*.

`LocalScope`(*body*)**.** L'analisi semantica della creazione di uno scope locale effettua ricorsivamente l'analisi semantica del corpo dello scope. Definendo ρ come risultato di questa analisi semantica, facciamo in modo che eventuali variabili locali dichiarate all'interno del corpo non siano più visibili all'esterno dello scope. Per esempio, nel comando `{ int a; a := 5 }` la variabile `a` non è più visibile dopo la parentesi graffa di chiusura.

`MethodCallCommand`(*receiver*, *name*, *actuals*)**.** L'analisi semantica del comando di invocazione di metodo è estremamente simile a quella dell'espressioni di invocazione di metodo in Figura 5.11. L'unica differenza è che qui non imponiamo alcun vincolo sul tipo di ritorno del metodo, che può quindi anche essere `void`.

$c_1; c_2$. L'analisi semantica della sequenza di comandi si richiama ricorsivamente sui due comandi, usando l'ambiente risultante dall'analisi semantica del primo per effettuare l'analisi semantica del secondo. In questo modo eventuali variabili locali dichiarate in c_1 possono essere usate da c_2.

5.4.1 L'implementazione dell'analisi semantica dei comandi

Dal momento che l'analisi semantica di un comando restituisce un ambiente, implementiamo il metodo di analisi semantica dentro `absyn/Command.java` come

```
private TypeChecker checker;

public final TypeChecker typeCheck(TypeChecker checker) {
  checker = typeCheck$0(this.checker = checker);
  if (next != null) return next.typeCheck(checker);
  else return checker;
}

protected abstract TypeChecker typeCheck$0(TypeChecker checker);
```

Il metodo `public` e `final` di nome `typeCheck()` effettua la parte di analisi semantica comune a tutti i comandi, che consiste nel chiamare il metodo ausiliario `typeCheck$0()`, annotare il type-checker risultante dall'analisi e richiamarsi ricorsivamente sul comando seguente, se esiste. In questo modo si implementa l'analisi semantica della sequenza di comandi.

Il metodo `typeCheck$0()` effettua la parte di analisi semantica specifica a ciascun comando, secondo le regole della Figura 5.15. Per esempio, dentro `absyn/IfThenElse.java` lo definiamo come

```
protected TypeChecker typeCheck$0(TypeChecker checker) {
  condition.mustBeBoolean(checker);
  then.typeCheck(checker);
  else.typeCheck(checker);
  return checker;
}
```

Invece dentro `absyn/For.java` lo definiamo come

```
protected TypeChecker typeCheck$0(TypeChecker checker) {
  TypeChecker initChecker = initialisation.typeCheck(checker);
  condition.mustBeBoolean(initChecker);
  update.typeCheck(initChecker);
  body.typeCheck(initChecker);
  return checker;
}
```

```
public class TypeChecker {
  private Type returnType;    private Table env;
  private int varNum;         private ErrorMsg errorMsg;

  private TypeChecker(Type returnType, Table env, int varNum, ErrorMsg errorMsg) {
    this.returnType = returnType; this.env = env;
    this.varNum = varNum; this.errorMsg = errorMsg;
  }
  public TypeChecker(Type returnType, ErrorMsg errorMsg) {
    this(returnType,Table.EMPTY,0,errorMsg);
  }
  public TypeChecker setReturnType(Type returnType) {
    return new TypeChecker(returnType,env,varNum,errorMsg);
  }
  public Type getReturnType() { return returnType; }
  public TypeChecker putVar(Symbol var, Type type) {
    return new TypeChecker
      (returnType,env.put(var,new TypeAndNumber(type,varNum)),varNum + 1,errorMsg);
  }
  public Type getVar(Symbol var) {
    TypeAndNumber tan = (TypeAndNumber)env.get(var);
    if (tan != null) return tan.getType(); else return null;
  }
  public int getVarNum(Symbol var) {
    TypeAndNumber tan = (TypeAndNumber)env.get(var);
    if (tan != null) return tan.getNumber(); else return -1;
  }
}
```

Figura 5.16: La classe `semantical/TypeChecker.java` che implementa un type-checker.

Si noti in quest'ultimo esempio come l'ambiente (in effetti, il type-checker) risultante dall'analisi di `initialisation` sia poi usato per effettuare l'analisi semantica di `condition`, `update` e `body`, conformemente alla Figura 5.15.

La Figura 5.16 mostra una revisione del type-checker della Figura 5.14. Adesso esso conosce il tipo di ritorno del metodo che si sta analizzando, fornito al momento della costruzione del type-checker tramite l'unico costruttore pubblico in Figura 5.16 e usato poi per implementare l'analisi semantica dei comandi `return`: in `absyn/Return.java` definiamo infatti:

```
  protected TypeChecker typeCheck$0(TypeChecker checker) {
    Type expectedReturnType = checker.getReturnType();
    if (returned == null && expectedReturnType != Type.VOID)
      error("missing return value");
    if (returned != null &&
```

```
        !returned.typeCheck(checker).canBeAssignedTo(expectedReturnType))
      error("illegal return type: " + expectedReturnType + " expected");
    return checker;
  }
```

conformemente alla Figura 5.15.

Si noti che il type-checker in Figura 5.16 associa alle variabile dell'ambiente non solo il loro tipo di dichiarazione, ma anche un numero progressivo, che sarà utile in fase di generazione del codice (Capitolo 6).

5.5 L'analisi semantica delle classi Kitten

Fare l'analisi semantica di una classe Kitten significa effettuare l'analisi semantica dei suoi membri, cioè campi, costruttori e metodi. Nulla va controllato per quanto riguarda i campi. Per quanto riguarda costruttori e metodi, invece, occorre effettuare l'analisi semantica del loro corpo. Essendo il loro corpo un comando, possiamo usare a tal fine le regole della Figura 5.15, cominciando l'analisi da un ambiente iniziale in cui i parametri del costruttore o del metodo sono legati al loro tipo di dichiarazione (incluso il parametro implicito `this`). A tal fine, definiamo una funzione che aggiunge a un ambiente una lista di variabili dichiarate come parametri formali:

$$\rho + \texttt{null} = \rho$$
$$\rho + \texttt{FormalParameters}(\mathit{type}, \mathit{name}, \mathit{next}) = (\rho + \mathit{next})[\mathit{name} \mapsto \tau[\![\mathit{type}]\!]]$$

L'analisi semantica di un costruttore o metodo con parametri formali *formals* e dichiarato in una classe il cui tipo semantico è κ viene quindi effettuata a partire da un ambiente iniziale

$$\overline{\rho} = [\texttt{this} \mapsto \kappa] + \mathit{formals}$$

Se *body* è il corpo del costruttore o metodo, si tratterà di verificare che il giudizio $\overline{\rho} : \mathit{body} : \rho'$ sia valido per un qualche ρ'.

Anche il metodo che fa l'analisi semantica dei membri di una classe si chiama `typeCheck()`. Esso è definito dentro `absyn/ClassMemberDeclaration.java` come

```
final void typeCheck(ClassType currentClass) {
  typeCheck$0(currentClass);
  if (next != null) next.typeCheck(currentClass);
}

protected abstract void typeCheck$0(ClassType currentClass);
```

ovvero tramite il solito metodo `final` che richiama, su tutta la lista dei membri della classe, il metodo ausiliario `typeCheck$0()` che effettua l'analisi specifica a ciascun membro. Abbiamo detto che l'analisi semantica dei campi non richiede nessun controllo: dentro la classe di sintassi astratta `absyn/FieldDeclaration.java` definiamo quindi:

```
protected void typeCheck$0(ClassType currentClass) {}
```

In `absyn/ConstructorDeclaration.java` definiamo invece:

```
protected void typeCheck$0(ClassType currentClass) {
  TypeChecker checker
    = new TypeChecker(Type.VOID,currentClass.getErrorMsg());
  checker = checker.putVar(Symbol.THIS,currentClass);
  if (getFormals() != null) checker = getFormals().typeCheck(checker);
  getBody().typeCheck(checker);
  getBody().checkForDeadcode();
}
```

Questo metodo comincia col costruire un type-checker con ambiente vuoto e che si aspetta come tipo di ritorno `Type.VOID`. Quindi aggiunge la variabile `this` legata al tipo semantico della classe e i parametri formali legati al loro tipo di dichiarazione, rispecchiando la definizione precedente di $\overline{\rho}$. Infine effettua il type-checking del corpo del costruttore e controlla che al suo interno non ci sia del codice morto (Sezione 4.3). Il ragionamento è simile nel caso della dichiarazione di un metodo, ma si usa il tipo di ritorno del metodo al posto di `Type.VOID` e si controlla che se il metodo ne ridefinisce un altro di una superclasse allora la ridefinizione del tipo di ritorno soddisfi il test `canBeAssignedToSpecial()` visto in Sezione 5.1. Se inoltre il metodo non ritorna `void`, si impone che il valore di ritorno del metodo `checkForDeadcode()` sia `true`, in modo da garantire che il metodo termini sempre con un comando `return`.

L'analisi semantica di Kitten descritta in questo capitolo è un po' semplificata rispetto alla realtà. In particolare non abbiamo considerato come dall'analisi della classe di partenza (Sezione 5.5) si arrivi a quella delle altre classi a cui essa fa riferimento. Questo è ottenuto facendo in modo che le regole delle Figure 5.10, 5.11 e 5.15, quando hanno bisogno di ottenere il tipo semantico delle espressioni di tipo, richiamino ricorsivamente il type-checking su tutte le classi che vi occorrono. Al fine di evitare cicli, si usa un flag `typeChecked` all'interno di `types.ClassType`.

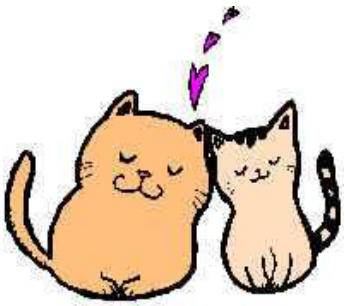
Esercizio 22. Si aggiunga alle espressioni la sintassi astratta di un'*espressione condizionale* $exp_1 ? exp_2 : exp_3$ che restituisce il valore di exp_2 se exp_1 è vera e il valore di exp_3 altrimenti. Si dia la sua regola di type-checking.

Esercizio 23. Si aggiunga ai comandi la sintassi astratta di un comando `switch`. Non ci si limiti a espressioni costanti nei vari casi. Si dia la regola di type-checking per tale comando.

Esercizio 24. Si aggiunga ai comandi la sintassi astratta dei comandi `break` e `continue`. Si diano le loro regole di type-checking, che devono garantire che tali comandi occorrano solo dentro un ciclo. Come modifichereste il type-checker in Figura 5.16 in modo da implementare tali controlli?

Capitolo 6

Generazione del Bytecode Kitten



L'analisi semantica del Capitolo 5 ha garantito che il codice Kitten non contiene alcun errore semantico. Ha inoltre annotato l'albero di sintassi astratta con informazioni relative al tipo statico delle espressioni che vi occorrono; gli accessi a campi, costruttori e metodi con la dichiarazione di campo, costruttore o metodo a cui fanno riferimento. Siamo ora nelle condizioni di generare del codice *intermedio*, cioè indipendente dall'architettura verso la quale stiamo compilando, ma pensato piuttosto per essere facilmente sintetizzabile a partire dall'albero di sintassi astratta e facilmente ottimizzabile. Esso verrà poi traslato in codice oggetto, specifico all'architettura verso cui compiliamo. Il codice intermedio che useremo è il *bytecode Kitten*, che può essere visto come una versione semplificata ed esplicitamente tipata del *Java bytecode*.

6.1 Il bytecode Kitten

Il bytecode Kitten è un linguaggio di programmazione pensato per essere eseguito da una macchina astratta che ha a disposizione:

1. un insieme di *variabili locali*, potenzialmente illimitato, che possono contenere valori primitivi o riferimenti ad oggetti o array;

2. uno stack di variabili temporanee, detto *stack degli operandi*, potenzialmente illimitato, che può contenere valori primitivi o riferimenti a oggetti o array;

```
Led():
  return void

on():
  load 0 of type Led
  const true
  putfield Led.state
  return void

off():
  load 0 of type Led
  const false
  putfield Led.state
  return void

isOn():
  load 0 of type Led
  getfield Led.state
  return boolean

isOff():
  load 0 of type Led
  getfield Led.state
  neg boolean
  return boolean
```

Figura 6.1: La compilazione in bytecode Kitten dei metodi della classe `Led` in Figura 1.4.

3. uno *stack di attivazione*, formato da un numero potenzialmente illimitato di *frame di attivazione* di metodi. Ciascun frame di attivazione contiene le variabili locali e lo stack di attivazione di un metodo;

4. una *memoria* o *heap*, che contiene oggetti e array allocati dinamicamente dal programma in esecuzione.

La maggior parte delle istruzioni del bytecode Kitten operano sulle variabili locali e sullo stack degli operandi. Un numero limitato (invocazione e ritorno da metodo) operano anche sullo stack di attivazione. Le sole operazioni che operano sulla memoria sono quelle di creazione di oggetto o array e di accesso a campi o array.

Si consideri la Figura 6.1. Essa mostra la compilazione in bytecode Kitten dei metodi della classe `Led` in Figura 1.4. All'inizio dell'esecuzione di un metodo o costruttore, lo stack degli operandi è vuoto e le variabili locali contengono i parametri attuali del metodo o costruttore. In particolare, la variabile locale numero 0 contiene sempre il riferimento all'oggetto corrente, cioè quello che nel codice sorgente sarebbe stato `this`, che è un parametro implicito in tutte le chiamate di metodo o costruttore. La variabile locale 1 contiene il primo parametro attuale esplicito, la variabile locale 2 il secondo parametro attuale esplicito, e così via. Si noti comunque che le variabili locali possono essere usate anche per contenere vere e proprie variabili locali ai metodi e non solo per contenere i parametri attuali. Nell'esempio in Figura 6.1, solo la variabile locale 0 è utilizzata, dal momento che nessun metodo richiede dei parametri espliciti né variabili locali. L'istruzione `load 0 of type Led` indica di copiare il riferimento all'oggetto corrente in cima allo stack degli operandi. L'istruzione `const` serve invece a caricare in cima allo stack degli operandi una costante. Nella Figura 6.1 si tratta di una costante booleana. Le istruzioni `getfield` e `putfield` servono, rispettivamente, a leggere e a scrivere un campo di un oggetto.

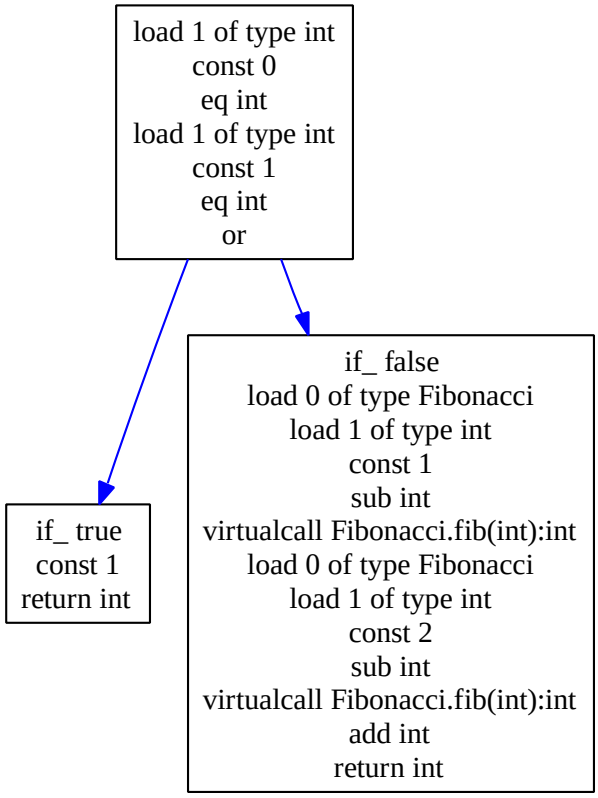


Figura 6.2: La compilazione in bytecode Kitten del metodo `fib()` in Figura 1.2.

L'istruzione `neg` nega il valore che sta in cima allo stack degli operandi. L'istruzione `return` termina l'esecuzione di un metodo o costruttore ritornando possibilmente un valore al chiamante.

Il bytecode in Figura 6.1 ha una struttura di controllo particolarmente semplice, dal momento che non prevede condizionali né cicli. La Figura 6.2 mostra un esempio più complesso: la compilazione in bytecode Kitten del metodo `fib()` in Figura 1.2. La presenza di un comando condizionale in Figura 1.2 diventa un'alternativa di controllo nel bytecode Kitten in Figura 6.2: il risultato dell'istruzione `or` determina l'instradamento del controllo verso il ramo `if_true` o verso quello `if_false`.

L'esempio precedente mostra che il bytecode Kitten è in effetti un grafo di *blocchi di codice* che contengono codice sequenziale. Un ulteriore esempio è la compilazione del ciclo:

```
for (int i := 0; i < 5; i := i + 1) {}
```

mostrata in Figura 6.3. Questa volta l'instradamento del controllo dipende dal risultato di un confronto. In particolare, il confronto fra la variabile locale 1, che contiene la variabile `i` del ciclo, e la costante intera 5 determina l'instradamento del codice verso il ramo `if_cmplt` (*IF the CoMParison is Less Than*) o verso quello `if_cmpge` (*IF the CoMParison is Greater than or Equal*).

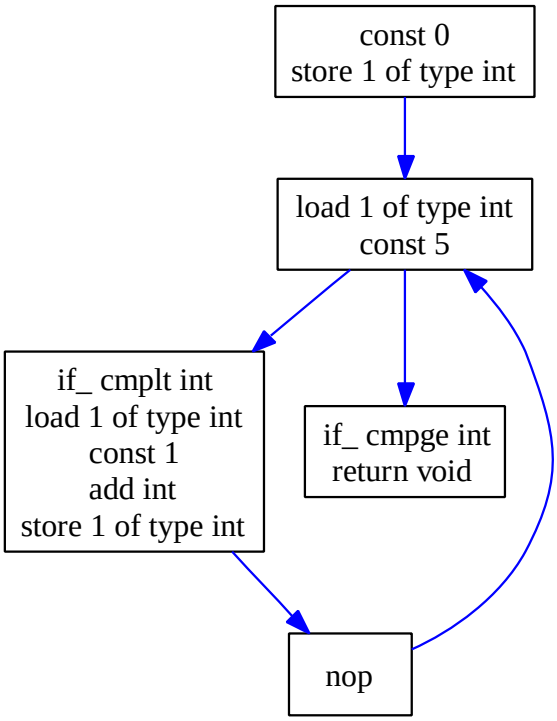


Figura 6.3: La compilazione in bytecode Kitten di un ciclo `for`.

6.1.1 Le istruzioni sequenziali

Esaminiamo adesso il set di istruzioni messe a disposizione dal bytecode Kitten. Per ognuna di esse mostriamo il suo effetto sulle variabili locali, sullo stack degli operandi e sulla memoria o heap. Dal momento che poche istruzioni operano sullo heap, lo indicheremo solo per quelle poche istruzioni per cui esso è effettivamente significativo. Le istruzioni del bytecode Kitten sono *tipate*, nel senso che è specificato il tipo degli operandi su cui possono operare. Esse non effettuano mai una promozione di tipo, per cui quando nella loro descrizione useremo il termine *sottotipo*, esso va inteso nel senso dell'operazione `canBeAssignedToSpecial()` della Sezione 5.1.

`nop`**.** Questa istruzione non modifica in nulla lo stato della macchina astratta. L'effetto della sua esecuzione può quindi essere rappresentato come in Figura 6.4.

`pop` *t***.** Rimuove la cima dello stack degli operandi, che deve avere tipo *t* (Figura 6.4).

`dup` *t***.** Duplica il valore in cima allo stack (Figura 6.4) che deve avere tipo *t*. Si noti che se tale valore fosse un riferimento a un oggetto o a un array allora verrebbe duplicato il riferimento, non l'oggetto o l'array.

`const` *value***.** Carica in cima allo stack un valore costante (Figura 6.5). È possibile caricare valori booleani, interi, float e la costante `nil`.

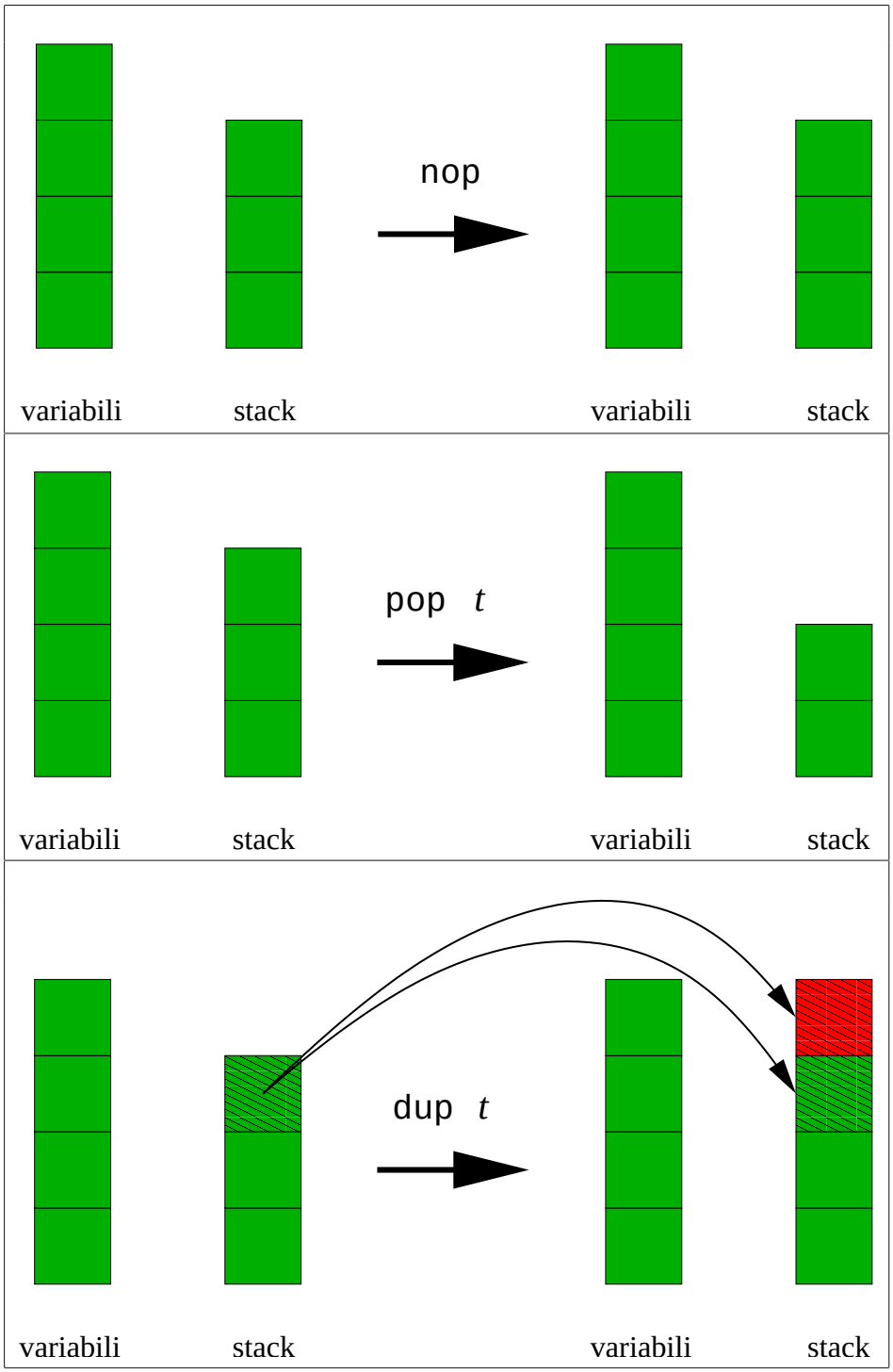


Figura 6.4: Le istruzioni nop, pop e dup del bytecode Kitten.

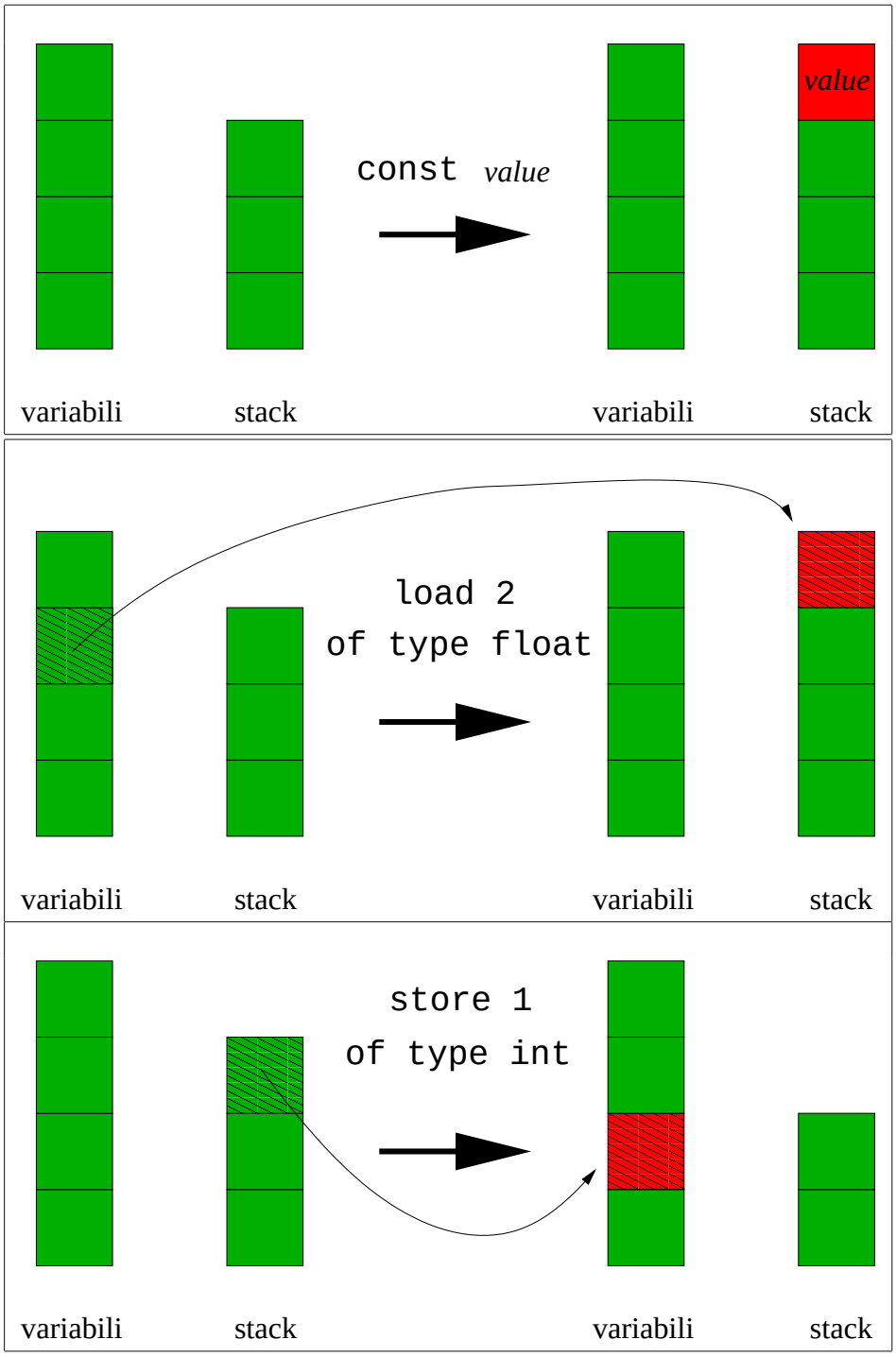


Figura 6.5: Le istruzioni `const`, `load` e `store` del bytecode Kitten.

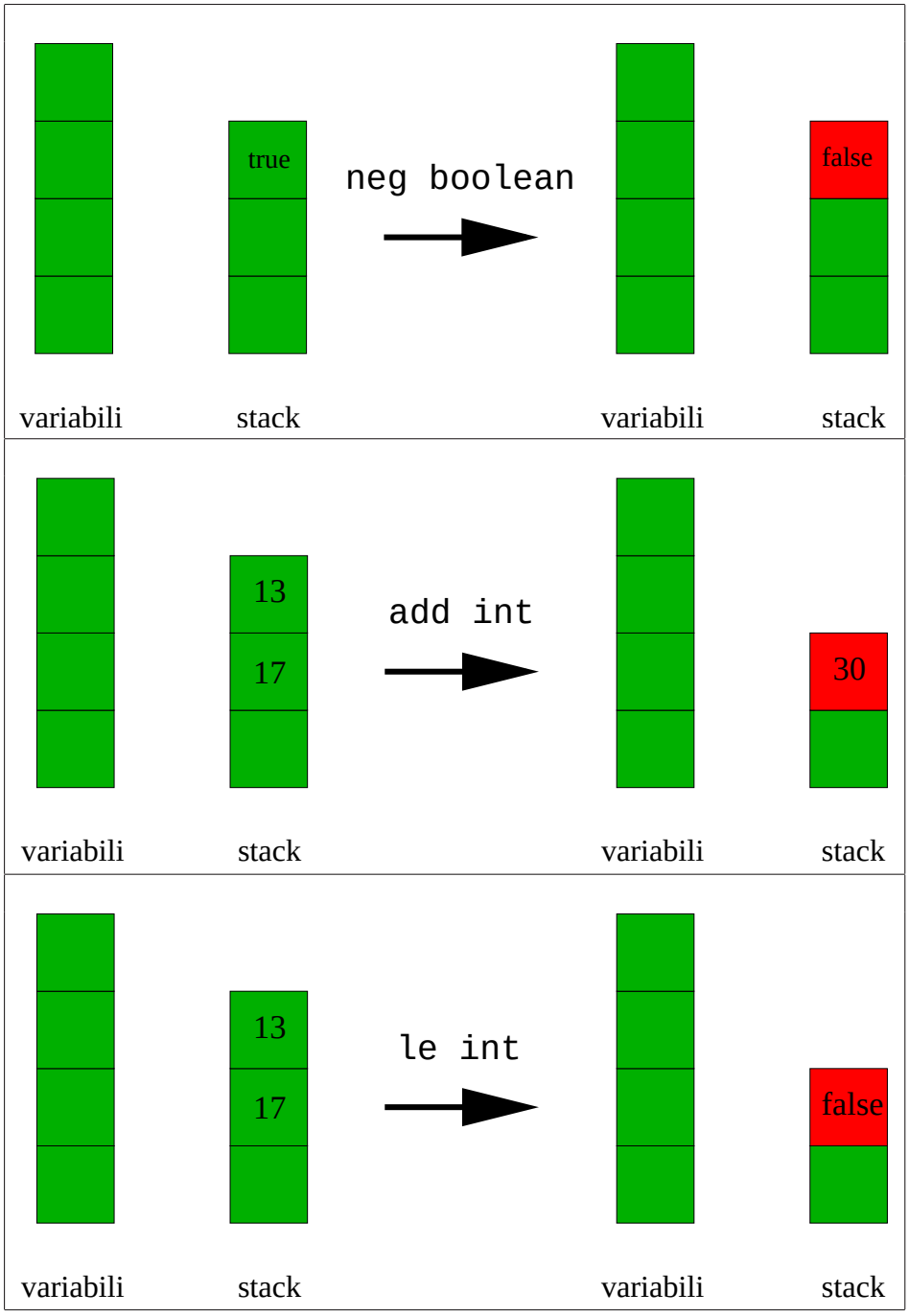


Figura 6.6: Le istruzioni `neg`, `add` e `le` del bytecode Kitten.

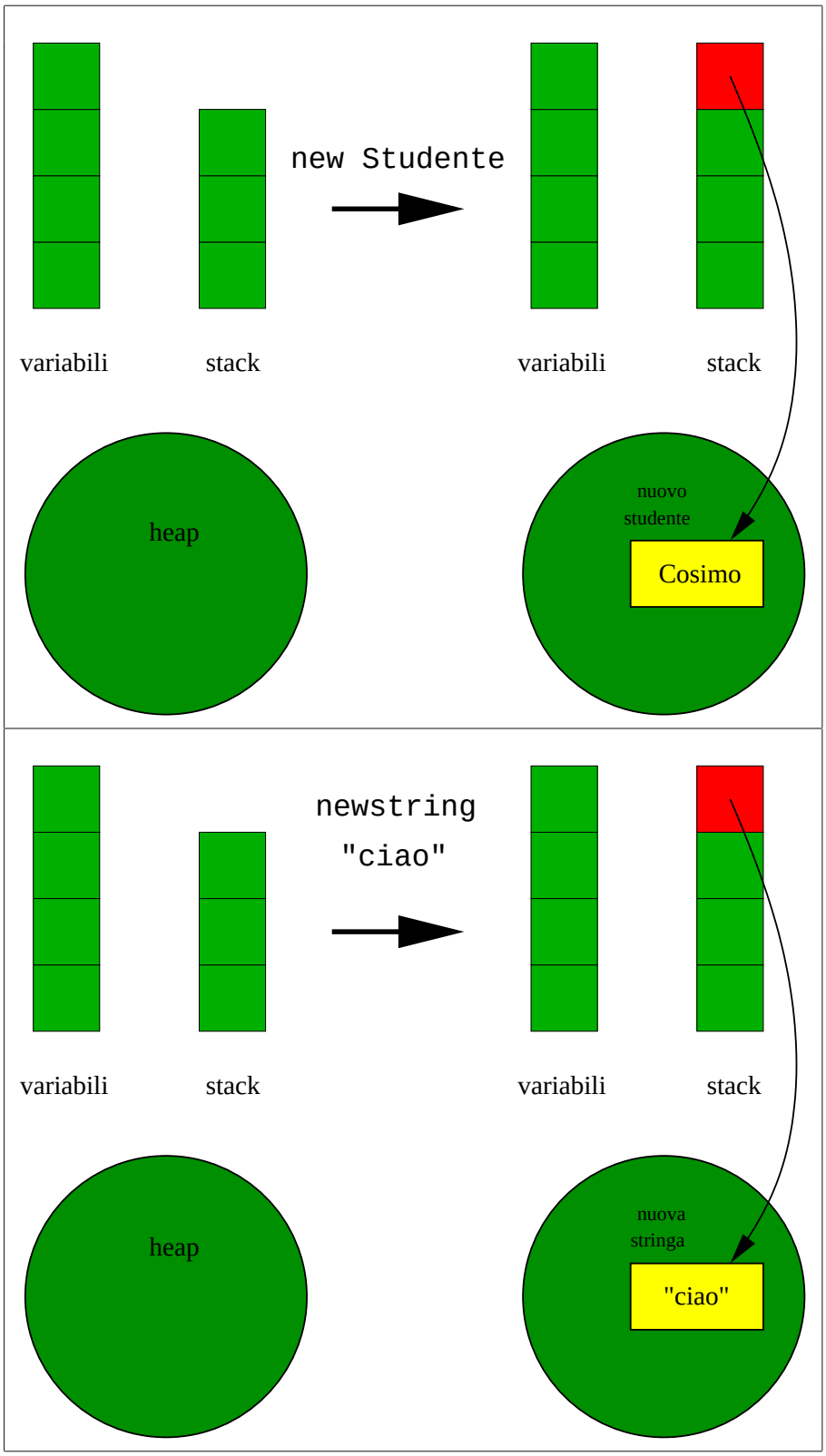


Figura 6.7: Le istruzioni `new` e `newstring` del bytecode Kitten.

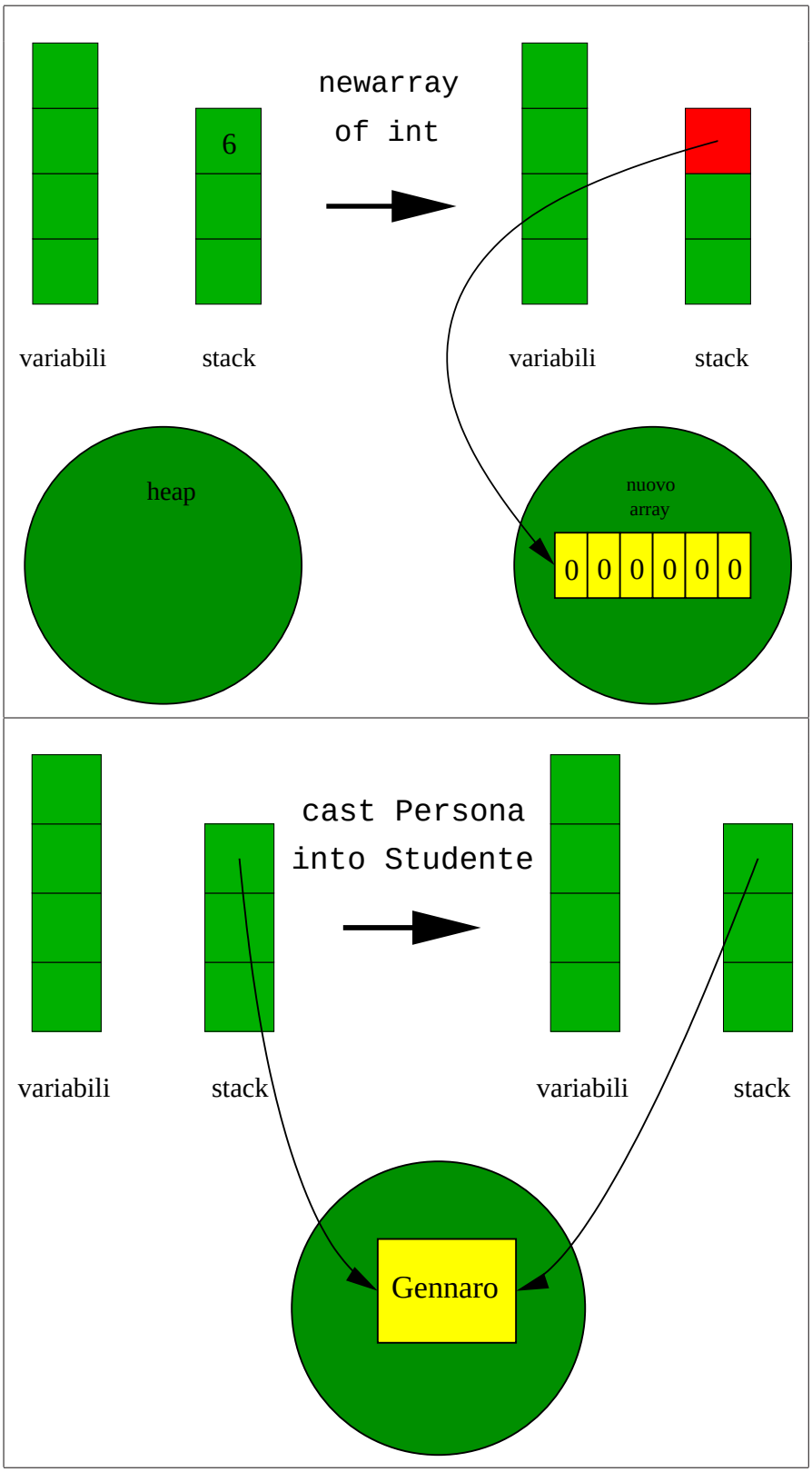


Figura 6.8: Le istruzioni `newarray` e `cast` del bytecode Kitten.

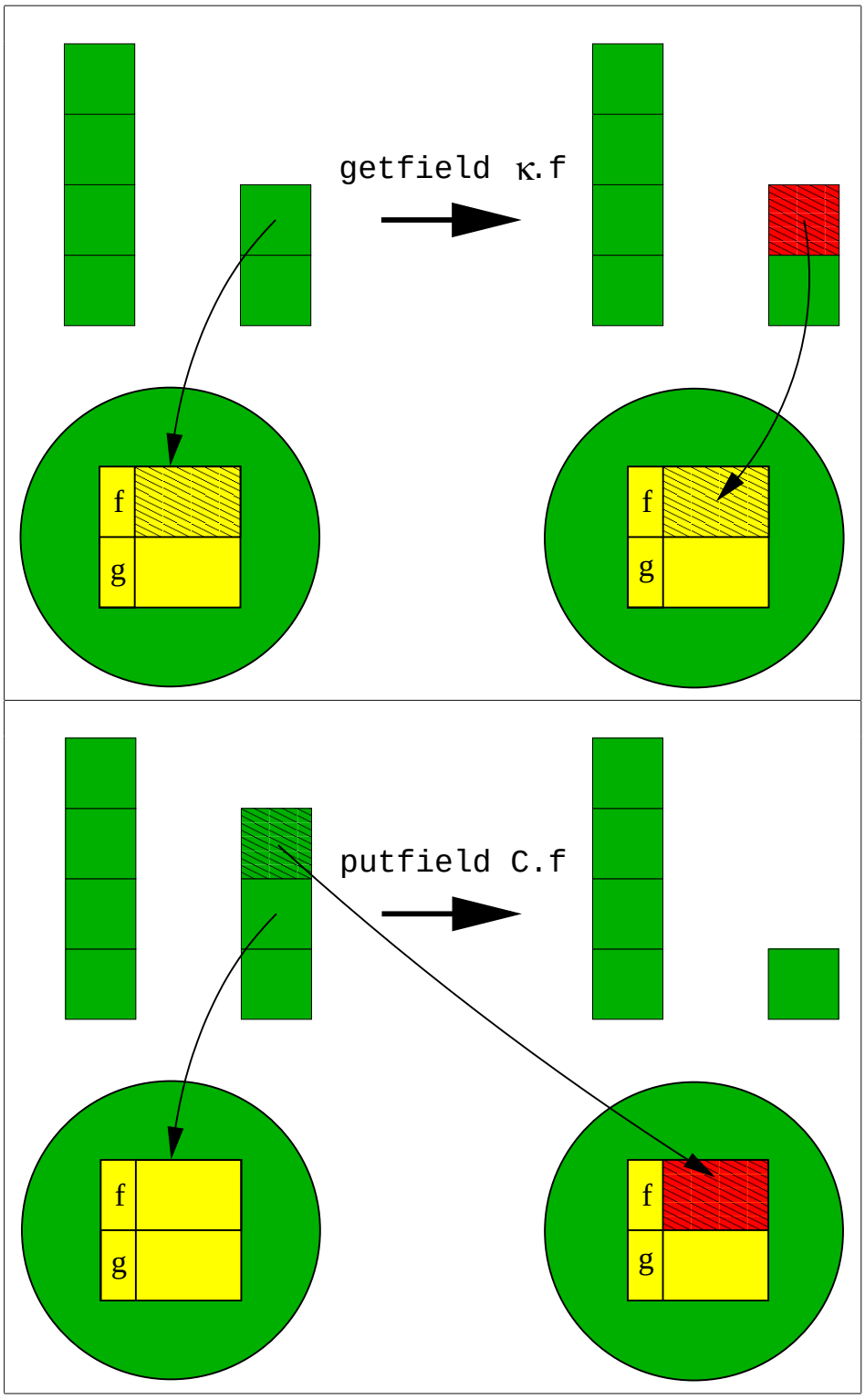


Figura 6.9: Le istruzioni `getfield` e `putfield` del bytecode Kitten.

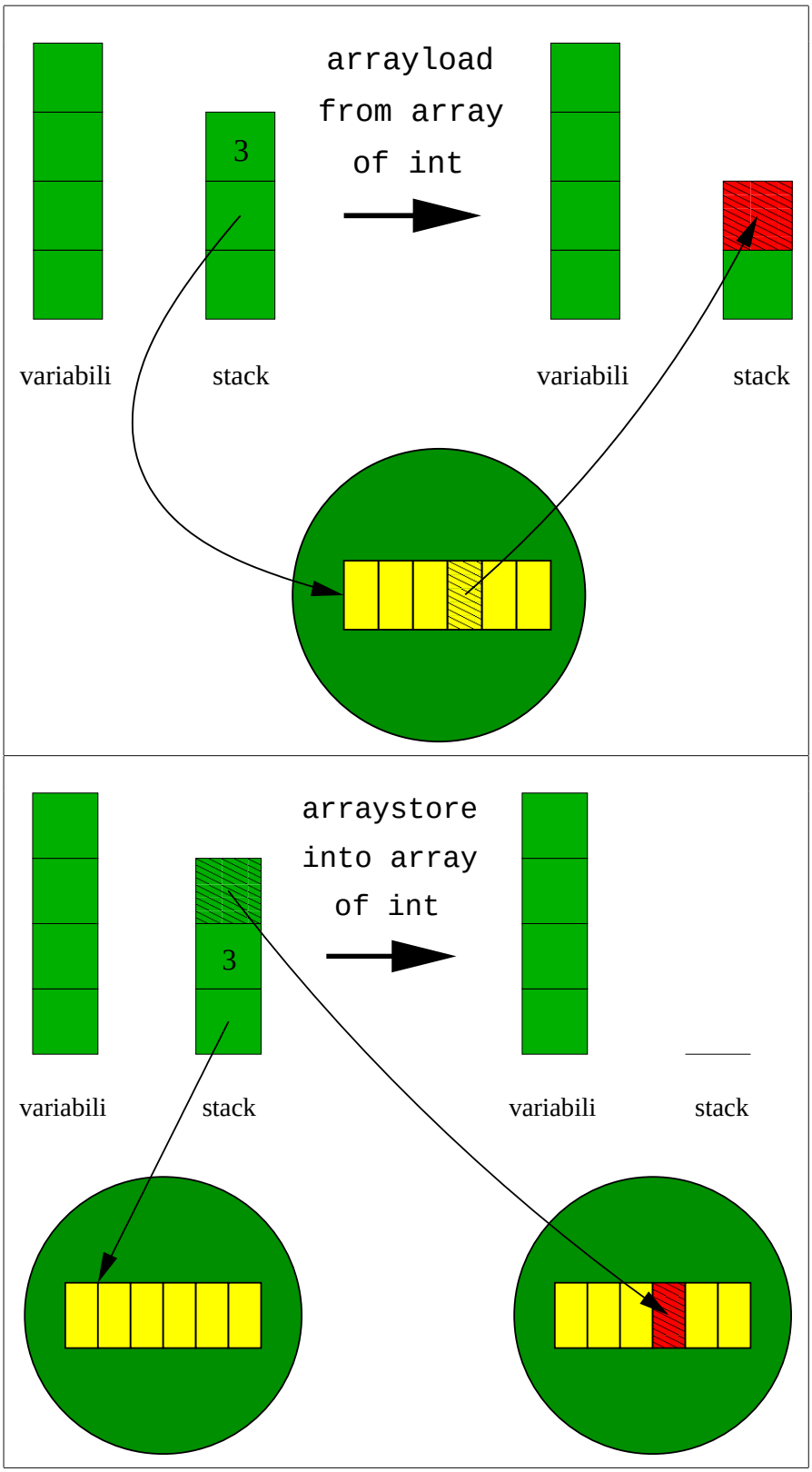


Figura 6.10: Le istruzioni `arrayload` e `arraystore` del bytecode Kitten.

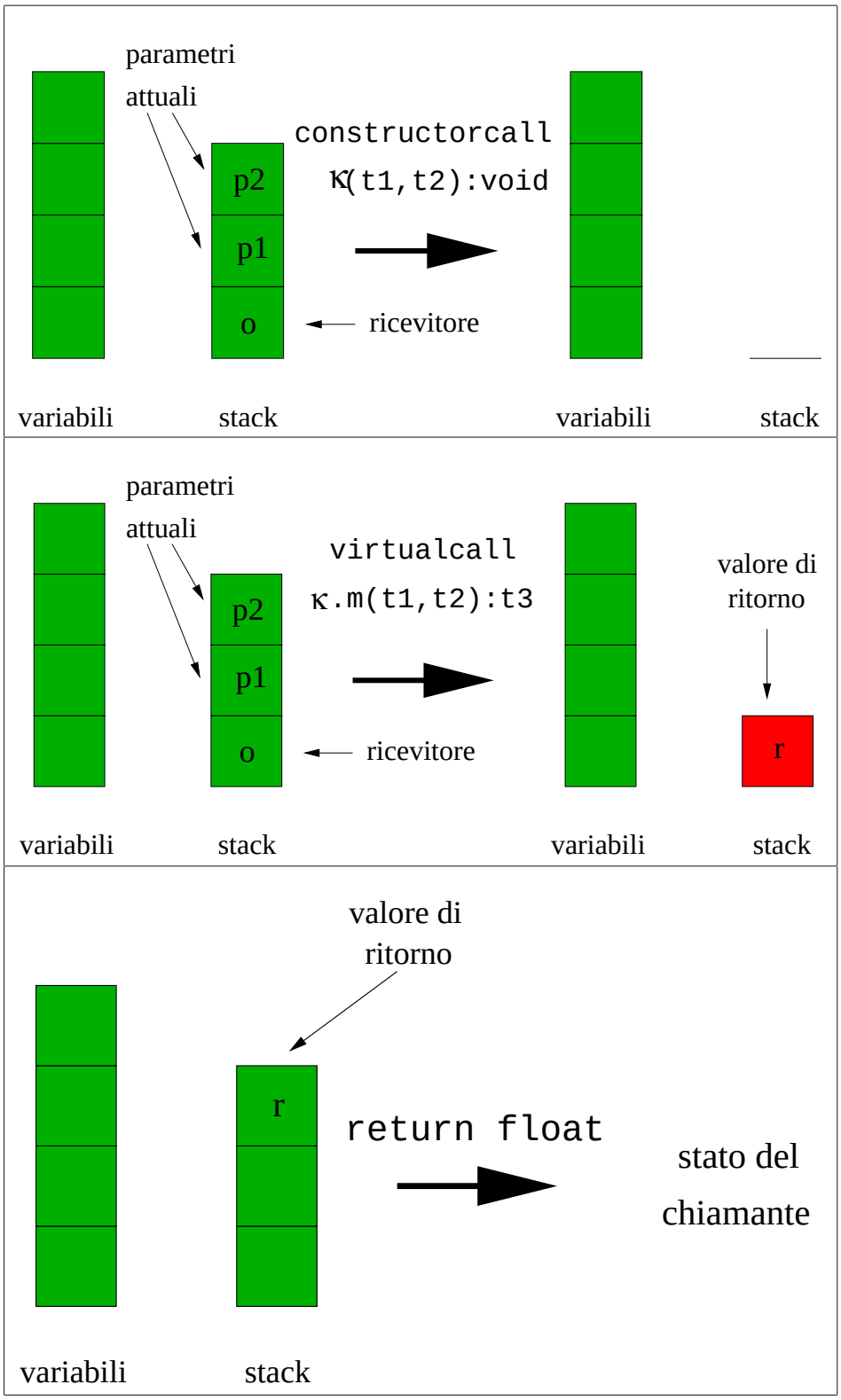


Figura 6.11: Le istruzioni `constructorcall`, `virtualcall` e `return` del bytecode Kitten.

`load` *l* `of type` *t*. Carica in cima allo stack degli operandi una copia del valore della variabile locale numero *l*, che deve contenere un valore di tipo *t* (Figura 6.5).

`store` *l* `of type` *t*. Sposta dentro la variabile locale numero *l* il valore che si trova in cima allo stack degli operandi. La cima di tale stack deve contenere un valore del tipo *t* e viene rimossa dall'operazione (Figura 6.5).

`neg` *t*. Nega il valore in cima allo stack degli operandi (Figura 6.6). Tale valore deve essere di tipo *t*. È possibile che *t* sia `boolean`, `int` o `float`. Si noti che il valore in cima allo stack che viene usato per calcolare l'operazione `neg` scompare dallo stack e viene sostituito dal risultato dell'operazione.

`add` *t*. Addiziona i due valori in cima allo stack degli operandi (Figura 6.6). Tali valori devono essere entrambi di tipo *t*. È possibile che *t* sia `int` o `float`. Similmente ad `add`, esistono anche le istruzioni `sub`, `mul` e `div`. Esistono anche le istruzioni `and` e `or` che però operano su due valori di tipo `boolean`. I due valori in cima allo stack che sono usati per calcolare l'operazione binaria scompaiono dallo stack e vengono sostituiti col risultato dell'operazione. Si noti che questa operazione non effettua alcuna promozione di tipo, per cui è vietato addizionare un intero con un numero in virgola mobile usando *t* = `float`.

`le` *t*. Controlla che il valore sotto la cima dello stack degli operandi sia minore o uguale al valore in cima allo stesso stack e sostituisce tali due valori con il risultato booleano del confronto. (Figura 6.6). I due valori devono essere di tipo *t* pari a `int` o `float`. Esistono anche le istruzioni `lt`, `ge` e `gt`. Infine esistono anche le istruzioni `eq` ed `ne`, che possono operare su valori di tipo *t* arbitrario, anche riferimento.

`new` κ. Crea un nuovo oggetto di classe κ. Un riferimento a tale oggetto viene posto in cima allo stack degli operandi (Figura 6.7). Si noti che non viene chiamato alcun costruttore per l'oggetto appena creato. Esso dovrà essere chiamato successivamente con un'esplicita istruzione `constructorcall` (si veda dopo).

`newstring` *s*. Crea un nuovo oggetto stringa che rappresenta *s* e pone in cima allo stack un riferimento all'oggetto, che è già inizializzato (Figura 6.7).

`newarray of` *t*. Crea un array i cui elementi hanno tipo *t*. La lunghezza dell'array è specificata in cima allo stack degli operandi ed è sostituita con un riferimento all'array appena creato (Figura 6.8).

`cast` t_1 *into* t_2. Effettua il cast del valore che sta in cima allo stack, che deve avere tipo t_1, nel tipo t_2. Questo bytecode può essere usato per fare cast verso il basso di tipi riferimento (nel qual caso un cast errato interrompe il programma) o per effettuare conversioni di tipo da `int` a `float` o viceversa (Figura 6.8).

`getfield` $\kappa.f$. Legge il campo f dell'oggetto il cui riferimento è in cima allo stack degli operandi. Tale riferimento viene rimosso e al suo posto viene messo il valore letto dal campo

(Figura 6.9). L'oggetto in cima allo stack deve essere di tipo κ o di una sottoclasse di κ. Se tale oggetto è `nil` il programma viene interrotto.

`putfield` $\kappa.f$**.** Scrive il valore in cima allo stack degli operandi dentro il campo f dell'oggetto il cui riferimento sta subito sotto la cima dello stack. I primi due elementi dello stack vengono rimossi (Figura 6.9). Il valore in cima allo stack deve essere del tipo del campo dentro cui si sta scrivendo o di un suo sottotipo. L'oggetto sotto la cima dello stack deve essere di tipo κ o di una sottoclasse di κ. Se tale oggetto è `nil` il programma viene interrotto.

`arrayload from array of` t**.** Copia in cima allo stack degli operandi il valore di un elemento di un array. L'indice dell'elemento è in cima allo stack. Subito sotto è presente il riferimento all'array (Figura 6.10) i cui elementi hanno tipo t o sottotipo di t. Se il riferimento all'array è `nil` o se l'indice è fuori dagli estremi dell'array, il programma viene interrotto. I primi due elementi in cima allo stack vengono rimossi dall'operazione e sostituiti con il valore letto dall'array.

`arraystore into array of` t**.** Scrive dentro a un array il valore che sta in cima allo stack. Sotto la cima dello stack c'è l'indice dell'elemento dell'array che deve essere scritto. Ancora sotto c'è il riferimento all'array che si sta modificando (Figura 6.10). Gli elementi dell'array che si sta modificando devono essere di tipo t o di un sottotipo di t. Se il riferimento all'array è `nil` o se l'indice è fuori dagli estremi dell'array, il programma viene interrotto. I primi tre elementi in cima allo stack vengono rimossi dall'operazione.

6.1.2 Le istruzioni di chiamata e ritorno da metodo

La Figura 6.11 mostra le istruzioni usate per chiamare un costruttore o metodo e per ritornare il controllo al chiamante. Esse operano come segue:

`constructorcall` $\kappa(\vec{t})$ `: void`**.** Chiama il costruttore della classe κ i cui parametri formali hanno tipo $\vec{t}$. I parametri attuali e l'oggetto che si sta inizializzando (cioè il *ricevitore* dal punto di vista del chiamante e il parametro implicito `this` di Kitten dal punto di vista del chiamato) sono passati tramite lo stack degli operandi e vengono rimossi alla fine della chiamata. Questo è mostrato in Figura 6.11, dal punto di vista del chiamante. La classe del ricevitore deve essere κ. Se il ricevitore è `nil` l'esecuzione del programma termina.

`virtualcall` $\kappa.m(\vec{t}) : t'$**.** Chiama il metodo di nome m e parametri formali di tipo $\vec{t}$ cercandolo a partire dalla classe del ricevitore e risalendo nella catena delle superclassi. Il ricevitore e i parametri attuali della chiamata si trovano sullo stack al momento della chiamata e vengono rimossi alla fine della chiamata e sostituiti con il valore di ritorno del metodo, nel caso in cui t' non è `void`. Questo è mostrato in Figura 6.11 dal punto di vista del chiamante. La classe del ricevitore deve essere κ o una sottoclasse di κ. Se il ricevitore è `nil` l'esecuzione del programma termina.

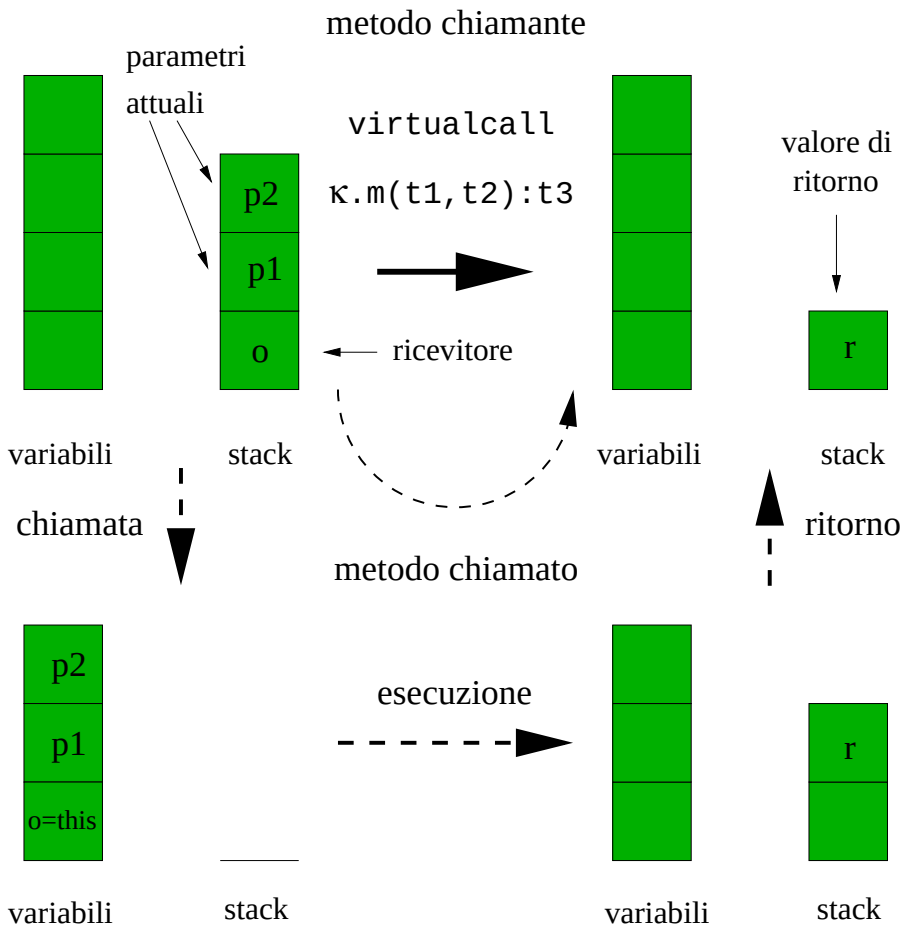


Figura 6.12: Il meccanismo di chiamata e ritorno da metodo.

`return` *t*. Termina l'esecuzione del metodo corrente, ritornando il controllo al chiamante, insieme a un eventuale valore di ritorno, che è la cima dello stack degli operandi (Figura 6.11) e deve avere tipo *t*.

Il funzionamento complessivo del meccanismo di chiamata e ritorno da costruttore o metodo è mostrato nella Figura 6.12. Il metodo chiamante prepara sullo stack degli operandi i parametri della chiamata, incluso il ricevitore della chiamata, indicato come *o* in Figura 6.12. Il metodo chiamato è esplicito nel caso della chiamata a un costruttore, mentre per le chiamate virtuali ai metodi è identificato sulla base della classe dell'oggetto a cui *o* fa riferimento. In entrambi i casi, esso inizia la sua esecuzione in un frame di attivazione nuovo, in cui le variabili locali contengono i parametri della chiamata e lo stack degli operandi è vuoto. Quando l'esecuzione del chiamato termina, se il metodo non ritorna `void` allora la cima dello stack degli operandi del chiamato contiene il valore di ritorno, *r* in Figura 6.12. La terminazione del metodo riabilita il frame di attivazione del chiamato, in cui però lo stack degli operandi è stato privato dei parametri e arricchito con il valore di ritorno *r* del metodo.

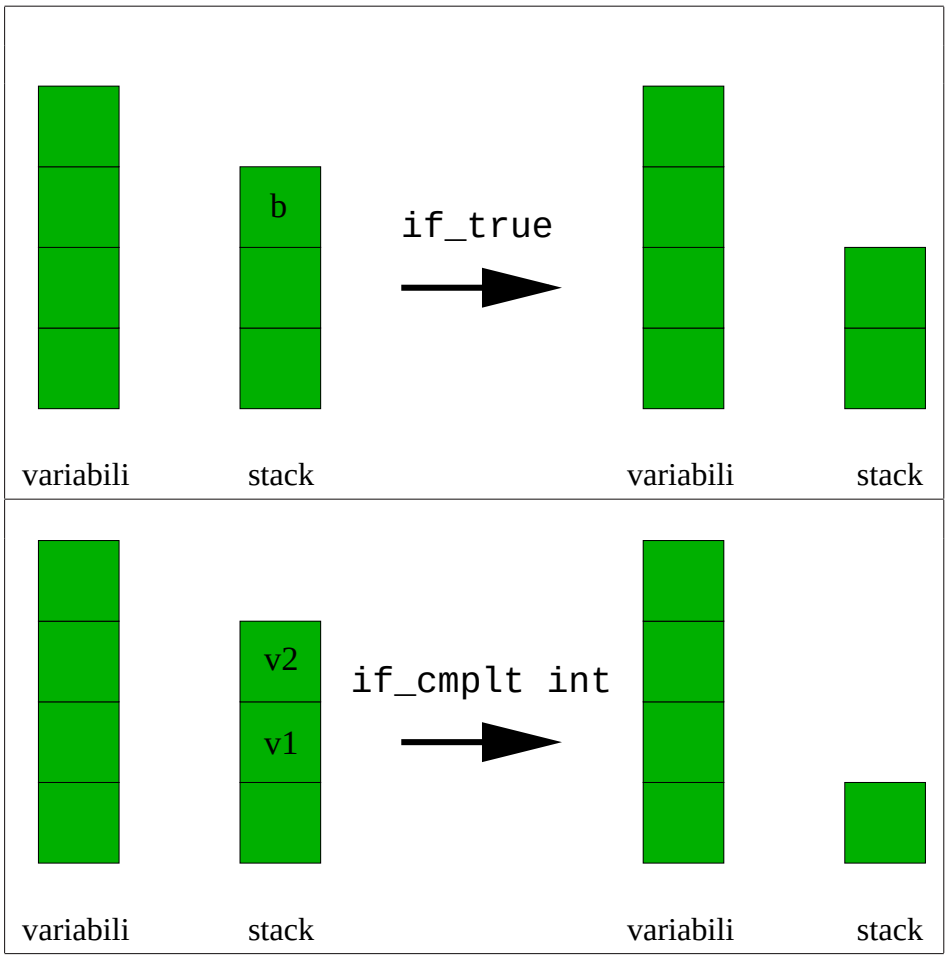


Figura 6.13: Le istruzioni `if_true` ed `if_cmplt` del bytecode Kitten.

6.1.3 Le istruzioni di diramazione

Le istruzioni di diramazione del bytecode Kitten sono sempre accoppiate all'inizio di due blocchi di codice con lo stesso predecessore. Esse indicano sotto quale condizione il controllo del programma deve essere instradato verso uno dei due blocchi. Ne sono esempi le istruzioni `if_true` ed `if_false` in Figura 6.1 e le istruzioni `if_cmplt int` e `if_cmpge int` in Figura 6.2. Quando la condizione espressa dall'istruzione condizionale è vera, essa viene eseguita, il che normalmente comporta l'eliminazione di alcuni valori dallo stack degli operandi.

Vediamo in dettaglio l'insieme delle istruzioni di diramazione del bytecode Kitten.

`if_true`**.** La condizione espressa da questa istruzione è che la cima dello stack degli operandi, che deve essere un booleano, sia il valore `true`. In tal caso il valore viene eliminato dallo stack (Figura 6.13). Esiste anche l'istruzione simmetrica `if_false`.

`if_cmplt` *t***.** La condizione espressa da questa istruzione è che l'elemento che sta sotto la cima dello stack degli operandi sia minore dell'elemento che sta in cima allo stack. Entrambi

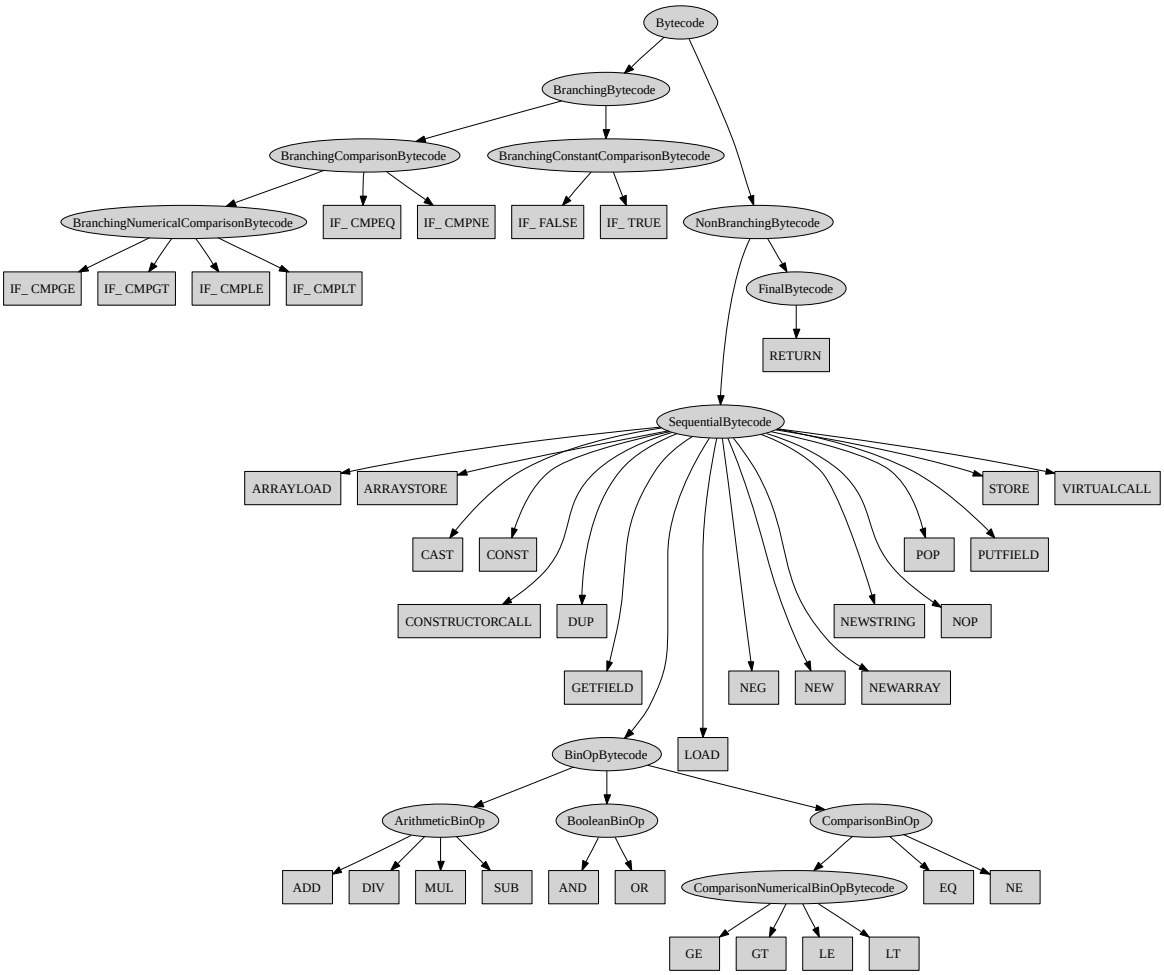


Figura 6.14: La gerarchia delle classi del package `bytecode` che rappresentano le istruzioni del bytecode Kitten. Le classi ovali sono classi astratte, quelle rettangolari sono classi concrete.

gli elementi devono avere tipo *t* e vengono rimossi dallo stack (Figura 6.13). Il tipo *t* può essere `int` o `float`. Esistono anche le istruzioni `if_cmple`, `if_cmpgt` ed `if_cmpge`. Esistono inoltre le istruzioni `if_cmpeq` ed `if_cmpne` la cui condizione, rispettivamente, è l'uguaglianza e la disuguaglianza dei due elementi in cima allo stack degli operandi. Queste ultime due istruzioni possono operare su tipi *t* arbitrari, anche riferimento.

6.1.4 L'implementazione del bytecode Kitten

Le istruzioni del bytecode Kitten che abbiamo descritto nelle sezioni precedenti sono implementate nel package `bytecode` come istanze della classe `bytecode/Bytecode.java`. La gerachia completa è mostrata in Figura 6.14. Le istruzioni vengono prima di tutto divise nelle due classi astratte `NonBranchingBytecode` e `BranchingBytecode`. La prima implementa le istruzioni

sequenziali delle Sezioni 6.1.1 e 6.1.2. La seconda implementa le istruzioni di diramazione della Sezione 6.1.3.

La creazione di un bytecode avviene tramite il suo costruttore, che richiede di specificare i tipi semantici su cui opera il bytecode. Per esempio, un'istruzione `arrayload from array of int` si crea con l'espressione Java

```
new ARRAYLOAD(Type.INT)
```

La classe `bytecode/BytecodeList.java` implementa poi una lista di bytecode che può essere inserita all'interno di un blocco di codice (come in Figura 6.2). La struttura dati che implementa tale blocco è la classe `translate/CodeBlock.java` il cui costruttore chiede di specificare la lista di bytecode contenuta nel blocco e la lista dei successori del blocco (eventualmente vuota).

Un metodo importante della classe dei bytecode sequenziali è `followedBy()`: esso richiede di specificare un blocco di codice e restituisce un blocco ottenuto aggiungendo il bytecode in testa al codice interno al blocco di codice. Per esempio, se il blocco di codice *b* contiene

```
const 1
return int
```

allora `new IF_TRUE().followedBy(b)` è un blocco di codice che contiene

```
if_true
const 1
return int
```

6.2 La generazione del bytecode Kitten per le espressioni

Mostriamo in questa sezione come tradurre la sintassi astratta di un'espressione Kitten in del bytecode Kitten.

Abbiamo visto che un programma scritto in bytecode Kitten è un insieme di blocchi all'interno dei quali si trova del codice, come mostrato in Figura 6.3. Il bytecode che genereremo per le espressioni sarà in effetti molto semplice, al punto che una sequenza di blocchi sarà sempre sufficiente per tutte le espressioni. Si noti comunque che questo non sarebbe più vero se Kitten ammettesse espressioni più complesse, come per esempio un'espressione condizionale come *exp* ? *exp* : *exp* (si veda l'Esercizio 25).

Ci sono tre contesti in cui un'espressione Kitten può trovarsi:

1. un contesto in cui di un'espressione serve il valore, come nel caso in cui essa occorre come lato destro di un assegnamento;

2. un contesto in cui di un'espressione serve sapere se è vera o falsa per decidere come instradare l'esecuzione del programma, come nel caso in cui essa occorre come test di un condizionale;

3. un contesto in cui il valore di un'espressione deve essere modificato, come nel caso in cui essa occorre alla sinistra di un assegnamento.

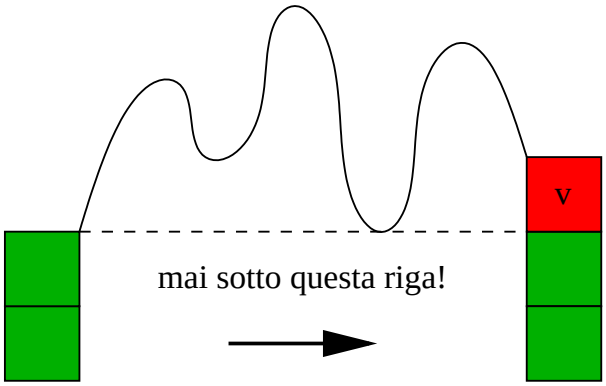


Figura 6.15: L'esecuzione del bytecode Kitten generato per un'espressione deve lasciare il valore dell'espressione sullo stack degli operandi e non deve modificare lo stack iniziale.

Compileremo un'espressione in tre modi diversi, sulla base del contesto in cui essa occorre. Tali modi sono detti rispettivamente *compilazione attiva*, *compilazione condizionale* e *compilazione passiva* dell'espressione. Descriviamo adesso in ordine questi tre tipi di compilazione delle espressioni.

6.2.1 La compilazione attiva delle espressioni

Quando di un'espressione ci interessa il valore, allora l'esecuzione del codice che vogliamo generare deve essere tale da:

1. lasciare intatti i valori iniziali sullo stack degli operandi;

2. aggiungere in cima allo stack degli operandi il valore dell'espressione.

Questi due principi sono mostrati in Figura 6.15. Il vincolo 1 è importante poiché esso ci permette di valutare in sequenza delle espressioni e ritrovarci alla fine i loro valori sullo stack. Questo è mostrato nella Figura 6.16, che mostra l'esecuzione del codice che genereremo per l'and logico di due espressioni e_1 ed e_2: prima generiamo del codice che valuta e_1 e ne lascia il valore sullo stack, poi del codice che valuta e_2 e ne lascia il valore sullo stack. Grazie al precedente vincolo 1, siamo certi che a questo punto il valore di e_1 è ancora nello stack, sotto la cima. Possiamo quindi aggiungere un bytecode `and` per ottenere il risultato cercato.

Se β è un blocco di codice, allora con la notazione $\boxed{ins} \rightarrow \beta$ rappresentiamo un blocco di codice al cui interno si trova l'istruzione (o le istruzioni) *ins* e che ha β come successore. La Figura 6.17 usa tale notazione per definire le regole per la generazione del codice per le espressioni Kitten. Esse sono formalizzate tramite una funzione $\gamma[\![\_]\!]$ che associa alla sintassi astratta delle espressioni il bytecode Kitten che ne calcola il valore e lo lascia in cima allo stack degli operandi. Tale funzione richiede in primo luogo di specificare l'espressione e di cui si vuole generare il bytecode. La notazione $\gamma[\![e]\!]$ è però ancora una funzione da `translate.CodeBlock`

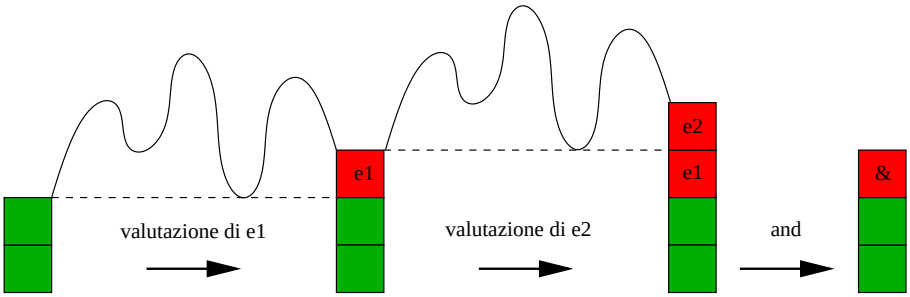


Figura 6.16: L'esecuzione del bytecode Kitten per l'and logico di due espressioni e_1 ed e_2.

in `translate.CodeBlock`, cioè la classe usata per rappresentare un blocco di bytecode (Sezione 6.1.4). In particolare, quello che occorre ancora specificare è il bytecode β che deve essere eseguito *dopo* la valutazione di e. Il bytecode $\gamma[\![e]\!](\beta)$ sarà quindi il bytecode che *prima* valuta l'espressione e, lasciandone il valore sullo stack degli operandi, e *dopo* esegue il codice β. Per esempio, la Figura 6.17 implica che

$$\gamma[\![\texttt{IntLiteral(3)}]\!](\boxed{\texttt{return int}}) = \boxed{\texttt{const 3}} \rightarrow \boxed{\texttt{return int}}$$

Questo modo di generare il codice si chiama *compilazione con continuazioni* e β è detta la *continuazione* della compilazione di e. La compilazione per continuazioni è molto elegante poiché permette di semplificare la fusione fra il codice generato per due parti sequenziali di un programma.

La Figura 6.17 usa la funzione $\gamma^{\tau}[\![e]\!]$ che rispetto a $\gamma[\![e]\!]$ effettua in più, se necessario, la promozione a τ del valore dell'espressione e. In Kitten essa è utile ogni qual volta si usa un valore intero in un punto in cui si richiedeva un valore in virgola mobile come per esempio nell'espressione `3 + 4.5`, in cui occorre convertire il valore intero 3 in un `float` prima di sommarlo con il valore 4.5. Quando potrebbe essere necessaria una promozione di tipo del valore di un'espressione, la Figura 6.17 compila l'espressione tramite $\gamma^{\tau}[\![\_]\!]$ piuttosto che tramite $\gamma[\![\_]\!]$. Lo stesso fenomeno lo incontreremo fra poco con i comandi, in un assegnamento del tipo:

```
float f := 13
```

dove l'intero 13 deve essere convertito in `float` prima dell'assegnamento. Tale funzione di conversione è definita a partire da $\gamma[\![e]\!]$:

$$\gamma^{\tau}[\![e]\!](\beta) = \begin{cases} \gamma[\![e]\!](\boxed{\texttt{cast from int into float}} \rightarrow \beta) & \text{se } \tau \text{ è } \texttt{float} \text{ ed } e \text{ ha tipo statico } \texttt{int} \\ \gamma[\![e]\!](\beta) & \text{altrimenti.} \end{cases} \tag{6.1}$$

Per esempio,

$$\begin{aligned} \gamma^{\texttt{int}}[\![\texttt{IntLiteral(3)}]\!](\boxed{\texttt{return int}}) &= \gamma[\![\texttt{IntLiteral(3)}]\!](\boxed{\texttt{return int}}) \\ &= \boxed{\texttt{const 3}} \rightarrow \boxed{\texttt{return int}} \end{aligned}$$

$$\gamma[\![\_]\!] : \texttt{absyn.Expression} \mapsto (\texttt{translate.CodeBlock} \mapsto \texttt{translate.CodeBlock})$$

$$\gamma[\![\texttt{Variable}(\mathit{name})]\!](\beta) = \boxed{\texttt{load}\ \mathit{num}\ \texttt{of type}\ \tau} \to \beta$$

dove *num* è il numero progressivo della variabile *name* nel metodo corrente

$$\gamma[\![\texttt{FieldAccess}(\mathit{receiver}, \mathit{name})]\!](\beta) = \gamma[\![\mathit{receiver}]\!]\left(\boxed{\texttt{getfield}\ \mathit{field}} \to \beta\right)$$

dove *field* è il campo identificato dall'analisi semantica (Figura 5.10)

$$\gamma[\![\texttt{ArrayAccess}(\mathit{array}, \mathit{index})]\!](\beta) = \gamma[\![\mathit{array}]\!]\left(\gamma[\![\mathit{index}]\!]\left(\boxed{\texttt{arrayload from array of}\ \tau} \to \beta\right)\right)$$

$$\gamma[\![\texttt{True}()]\!](\beta) = \boxed{\texttt{const true}} \to \beta \qquad \gamma[\![\texttt{False}()]\!](\beta) = \boxed{\texttt{const false}} \to \beta$$

$$\gamma[\![\texttt{IntLiteral}(\mathit{value})]\!](\beta) = \gamma[\![\texttt{FloatLiteral}(\mathit{value})]\!](\beta) = \boxed{\texttt{const}\ \mathit{value}} \to \beta$$

$$\gamma[\![\texttt{String}(\mathit{value})]\!](\beta) = \boxed{\texttt{newstring}\ \mathit{value}} \to \beta \qquad \gamma[\![\texttt{Nil}()]\!](\beta) = \boxed{\texttt{const nil}} \to \beta$$

$$\gamma[\![\texttt{NewObject}(\mathit{className}, \mathit{actuals})]\!](\beta) = \boxed{\begin{array}{l}\texttt{new}\ \kappa \\ \texttt{dup}\ \kappa\end{array}} \to \gamma^{\vec{t}}[\![\mathit{actuals}]\!]\left(\boxed{\texttt{constructorcall}\ \mathit{con}} \to \beta\right)$$

dove $\mathit{con} = \kappa.\kappa(\vec{t})$: `void` è il costruttore identificato dall'analisi semantica (Figura 5.11)

$$\gamma[\![\texttt{NewArray}(\mathit{elementsType}, \mathit{size})]\!](\beta) = \gamma[\![\mathit{size}]\!]\left(\boxed{\texttt{newarray of}\ \tau\texttt{.getElementsType()}} \to \beta\right)$$

$$\gamma[\![\texttt{MethodCallExpression}(\mathit{receiver}, \mathit{name}, \mathit{actuals})]\!] = \gamma[\![\mathit{receiver}]\!]\left(\gamma^{\vec{t}}[\![\mathit{actuals}]\!]\left(\boxed{\texttt{virtualcall}\ \mathit{method}} \to \beta\right)\right)$$

dove $\mathit{method} = \kappa.\mathtt{m}(\vec{t}) : t'$ è il metodo identificato dall'analisi semantica (Figura 5.11)

$$\gamma[\![\texttt{Not}(\mathit{expression})]\!](\beta) = \gamma[\![\texttt{Minus}(\mathit{expression})]\!](\beta) = \gamma[\![\mathit{expression}]\!]\left(\boxed{\texttt{neg}\ \tau} \to \beta\right)$$

$$\gamma[\![\texttt{Cast}(\mathit{type}, \mathit{expression})]\!](\beta) = \gamma[\![\mathit{expression}]\!]\left(\boxed{\texttt{cast from}\ \tau'\ \texttt{into}\ \tau} \to \beta\right) \quad \text{con } \tau' \text{ è tipo statico di } \mathit{expression}$$

$$\gamma[\![\texttt{And}(\mathit{left}, \mathit{right})]\!](\beta) = \gamma[\![\mathit{left}]\!]\left(\gamma[\![\mathit{right}]\!]\left(\boxed{\texttt{and}} \to \beta\right)\right)$$

$$\gamma[\![\texttt{Addition}(\mathit{left}, \mathit{right})]\!](\beta) = \gamma^{\tau}[\![\mathit{left}]\!]\left(\gamma^{\tau}[\![\mathit{right}]\!]\left(\boxed{\texttt{add}\ \tau} \to \beta\right)\right)$$

$$\gamma[\![\texttt{LessThanOrEqual}(\mathit{left}, \mathit{right})]\!](\beta) = \gamma^{\ell}[\![\mathit{left}]\!]\left(\gamma^{\ell}[\![\mathit{right}]\!]\left(\boxed{\texttt{le}\ \ell} \to \beta\right)\right) \quad \text{con } \ell \text{ minimo supertipo comune fra il tipo statico di } \mathit{left} \text{ e di } \mathit{right}$$

Figura 6.17: La funzione $\gamma[\![\_]\!]$ che genera il bytecode Kitten che valuta le espressioni. Il tipo τ è il tipo statico assegnato all'espressione durante la sua analisi semantica.

mentre

$$\begin{aligned}
&\gamma^{\texttt{float}}[\![\texttt{IntLiteral(3)}]\!](\boxed{\texttt{return float}}) \\
&= \gamma[\![\texttt{IntLiteral(3)}]\!](\boxed{\texttt{cast from int into float}} \to \boxed{\texttt{return float}}) \\
&= \boxed{\texttt{const 3}} \to \boxed{\texttt{cast from int into float}} \to \boxed{\texttt{return float.}}
\end{aligned}$$

L'esempio precedente sarebbe quello di un'istruzione `return 3` che occorre all'interno di un metodo il cui tipo di ritorno è `float`. La notazione $\gamma^{\tau}[\![\_]\!]$ viene infine estesa a sequenze di espressioni e di tipi (di uguale lunghezza), ottenendo la notazione $\gamma^{\vec{t}}[\![\_]\!]$, definita come segue:

$$\gamma^{\epsilon}[\![\texttt{null}]\!](\beta) = \beta$$

$$\gamma^{\tau::\vec{t}}[\![\texttt{ExpressionSeq}(\mathit{head}, \mathit{tail})]\!](\beta) = \gamma^{\tau}[\![\mathit{head}]\!](\gamma^{\vec{t}}[\![\mathit{tail}]\!](\beta)) \ .$$

Per esempio:

$$\begin{aligned}
&\gamma^{\texttt{float::float}}[\![\ \begin{array}{l}\texttt{ExpressionSeq(FloatLiteral(3.4),}\\ \texttt{ExpressionSeq(IntLiteral(4),null))}\end{array}\ ]\!](\boxed{\texttt{add float}})\\
&\quad = \gamma^{\texttt{float}}[\![\texttt{FloatLiteral(3.4)}]\!](\gamma^{\texttt{float}}[\![\texttt{IntLiteral(4)}]\!](\boxed{\texttt{add float}}))\\
&\quad = \gamma^{\texttt{float}}[\![\texttt{FloatLiteral(3.4)}]\!](\boxed{\texttt{const 4}} \to \boxed{\texttt{cast from int to float}} \to \boxed{\texttt{add float}})\\
&\quad = \boxed{\texttt{const 3.4}} \to \boxed{\texttt{const 4}} \to \boxed{\texttt{cast from int to float}} \to \boxed{\texttt{add float.}}
\end{aligned}$$

Commentiamo adesso le regole di generazione del bytecode Kitten in Figura 6.17.

`Variable`(*name*)**.** Per caricare sullo stack il valore di una variabile locale, usiamo il bytecode `load` della Figura 6.5. Il numero della variabile locale è già stato determinato in fase di analisi semantica e annotato dentro all'ambiente del punto di programma in cui ci troviamo, insieme al tipo della variabile (Figura 5.16).

`FieldAccess`(*receiver*, *name*)**.** Per accedere a un campo dell'oggetto o contenuto in *receiver*, generiamo inizialmente il bytecode che lascia sullo stack degli operandi il riferimento ad o. Questo è ottenuto richiamando ricorsivamente la generazione del bytecode per *receiver*. Come continuazione, gli passiamo un blocco che contiene un bytecode `getfield` (Figura 6.9) e che è legato alla continuazione β. L'effetto globale è quindi quello di valutare *receiver*, leggere il valore del campo di nome *name* e quindi continuare con la continuazione β. Si noti che il campo da leggere è già stato identificato in fase di analisi semantica (*field* in Figura 5.10).

`ArrayAccess`(*array*, *index*)**.** Leggere un elemento di un array richiede in primo luogo di valutare l'espressione che contiene il riferimento all'array. Questo è ottenuto compilando ricorsivamente *array*. Come continuazione gli diamo la compilazione di *index*, seguita dal bytecode `arrayload` e dalla continuazione β. Si noti che il tipo statico τ dell'array è già stato calcolato in fase di analisi semantica. Il bytecode `arrayload` consumerà dallo stack il riferimento all'array e l'indice da cui leggere e li sostituirà con il valore dell'elemento letto (Figura 6.10).

`True`(), `False`(), `IntLiteral`(*value*), `FloatLiteral`(*value*), `String`(*value*), `Nil`()**.** Dal momento che queste classi di sintassi astratta rappresentano delle costanti, usiamo il bytecode `const` della Figura 6.5 e `newstring` della Figura 6.7 per caricare tali costanti in cima allo stack.

`NewObject`(*className*, *actuals*)**.** Questo nodo di sintassi astratta per la creazione di un oggetto di classe *className* è stato annotato durante l'analisi semantica con il costruttore $con = \kappa.\kappa(\vec{t})$: `void` della classe κ che è il corrispondente semantico di *className*. Tale costruttore è il più specifico fra quelli che possono essere chiamati da questa espressione sulla base del tipo statico dei parametri attuali (Figura 5.11). Il codice che generiamo inizia con un bytecode `new` κ che crea un nuovo oggetto o di classe κ e ne pone in cima allo stack

un riferimento (Figura 6.7). Tale riferimento viene quindi duplicato dal bytecode `dup` κ (Figura 6.4). Segue la compilazione dei parametri attuali del costruttore. A questo punto sullo stack troviamo due copie di un riferimento ad o sormontate dai valori dei parametri attuali. Con il bytecode `constructorcall` otteniamo quindi di chiamare il costruttore legando `this` ad o e i parametri attuali ai parametri formali (Figura 6.11). Per esempio, la compilazione di

```
NewObject(className,
        ExpressionSeq(IntLiteral(3), ExpressionSeq(IntLiteral(4), null)))
```

è

```
new κ
dup κ
const 3
const 4
constructorcall con
```

seguita dalla continuazione β (nell'ipotesi che non serva promozione di tipo nel passaggio dei parametri interi al costruttore). Dalla Figura 6.11 sappiamo che il bytecode `constructorcall` rimuove dallo stack degli operandi sia i parametri attuali che o. Questo è il motivo per cui usiamo il bytecode `dup`: senza di esso il riferimento ad o andrebbe perso dallo stack e avremmo ottenuto di inizializzare un oggetto che subito dopo diventava irraggiungibile.

`NewArray`(*elementsType*, *size*)**.** Il bytecode generato per la creazione di un array inizia con la compilazione dell'espressione *size* che lascia in cima allo stack la dimensione richiesta per l'array. Basta quindi proseguire il codice con un bytecode `newarray` che consuma tale dimensione e la sostituisce con un riferimento a un nuovo array (Figura 6.8). Segue la continuazione β. Si ricordi che il tipo statico τ di questa espressione è il tipo dell'array che stiamo creando (Figura 5.11).

`MethodCallExpression`(*receiver*, *name*, *actuals*)**.** L'invocazione di un metodo è compilata in modo molto simile all'invocazione di un costruttore per un nodo `NewObject` di sintassi astratta (si veda sopra). La differenza è che si usa il bytecode `virtualcall` invece di `constructorcall` (Figura 6.11). Inoltre il riferimento all'oggetto ricevitore della chiamata è il valore lasciato sullo stack dal bytecode generato per *receiver*, piuttosto che un nuovo oggetto come per `NewObject`. Si ricordi che l'analisi semantica ha garantito che il metodo invocato ha un tipo di ritorno diverso da `void` (Figura 5.11). Siamo quindi sicuri che il bytecode `virtualcall` lascia sullo stack un valore di ritorno (Figura 6.11), che è il valore di questa espressione d'invocazione di un metodo.

`Not`(*expression*) **e** `Minus`(*expression*)**.** Entrambe queste espressioni sono compilate in del bytecode che inizia con la compilazione ricorsiva di *expression* e continua con il bytecode `neg` (Figura 6.6) e con la continuazione β. Si noti comunque che il tipo τ su cui opera `neg` è diverso: esso è `boolean` per `Not` ed è `int` oppure `float` per `Minus` (Figura 5.11).

`Cast`(*type*, *expression*)**.** La compilazione di un cast verso il basso è essenzialmente la compilazione dell'espressione di cui si sta facendo il cast, seguita dalla continuazione β. In più inseriamo un bytecode `cast` che effettua il cast o la conversione di tipo da `float` ad `int` (Figura 6.8) nel caso in cui il cast sia in effetti una richiesta di arrotondamento di un valore a virgola mobile (`3.14 as int`). Si noti la differenza fra queste due situazioni: la conversione da `float` ad `int` modifica la rappresentazione binaria del valore in cima allo stack ma non può mai fallire (sappiamo con certezza che in cima allo stack c'è un `float`). La verifica di tipo non effettua invece alcuna modifica sul valore in cima allo stack, ma può fallire bloccando l'esecuzione del programma.

`BinOp`(*left*, *right*)**.** La compilazione di un'operazione binaria è il codice formato dalla compilazione di *left* seguita dalla compilazione di *right* seguita da un bytecode che implementa l'operazione binaria opportuna e infine dalla continuazione β. Gli esempi mostrati in Figura 6.17 presentano tutte le tipologie di espressioni binarie. Quelle logiche usano un bytecode `and` od `or` per il quale non serve specificare il tipo degli operandi (è sempre `boolean`). Quelle aritmetiche possono invece operare sia su `int` che su `float` e i loro operandi potrebbero richiedere una promozione di tipo, per cui usiamo per essi $\gamma^{\tau}[\![\_]\!]$ piuttosto che $\gamma[\![\_]\!]$. Le operazioni binarie di confronto possono operare su tipo arbitrari e possono anch'esse richiedere una conversione di tipo per gli operandi.

Consideriamo adesso l'implementazione delle regole di compilazione in Figura 6.17. Un blocco di codice lo implementiamo come un oggetto di classe `translate.CodeBlock` contenente una lista di bytecode Kitten ed eventualmente legato ad altri blocchi successori. L'implementazione della funzione γ è ottenuta tramite i seguenti due metodi aggiunti ad `absyn/Expression.java`:

```
protected abstract CodeBlock translate(CodeBlock continuation);

public final CodeBlock translateAs(Type type, CodeBlock continuation) {
  if (staticType == Type.INT && type == Type.FLOAT)
    continuation = new CAST(Type.INT,Type.FLOAT).followedBy(continuation);
  else
    return translate(continuation);
}
```

Il primo implementa $\gamma[\![\_]\!]$ ed è lasciato `abstract`. Esso verrà istanziato nelle sottoclassi di `absyn.Expression` con l'implementazione delle regole in Figura 6.17. Il secondo implementa la funzione $\gamma^{\tau}[\![\_]\!]$ dell'Equazione 6.1. Si noti l'uso di `followedBy()` per aggiungere un bytecode in cima a un blocco di codice.

Mostriamo alcuni esempi di istanziazione del metodo `translate`(). In `absyn/True.java` definiamo

```
public final CodeBlock translate(CodeBlock continuation) {
  return new CONST(true).followedBy(continuation);
}
```

che rispecchia fedelmente quanto riportato in Figura 6.17.

In `absyn/Variable.java` defininiamo

```
public CodeBlock translate(CodeBlock continuation) {
  return new LOAD(getVarNum(),getStaticType()).followedBy(continuation);
}
```

Il numero della variabile era stato annotato in fase di analisi semantica (Sezione 5.3.1). Utilizziamo anche il tipo τ annotato per questa espressione, accessibile tramite `getStaticType()`. Ancora una volta, questa implementazione riflette la definizione in Figura 6.17.

Dentro `absyn/BinOp.java` definiamo

```
public final CodeBlock translate(CodeBlock continuation) {
  Type ell = getLeft().getStaticType()
    .leastCommonSupertype(getRight().getStaticType());
  return getLeft().translateAs
    (ell,getRight().translateAs
     (ell,operator(ell).followedBy(continuation)));
}

protected abstract BinOpBytecode operator(Type type);
```

L'idea è di calcolare il minimo sovratipo comune ℓ fra i tipi statici dei due operandi, compilarli entrambi con $\gamma^{\ell}[\![\_]\!]$ e farli quindi seguire da un bytecode binario specifico all'operazione binaria che si sta compilando. Tale bytecode è fornito dal metodo ausiliario `operator()` che è per esempio definito dentro `absyn/Addition.java` come

```
protected BinOpBytecode operator(Type type) {
  return new ADD((NumericalType)type);
}
```

Si noti che questo modo di procedere generalizza le tre ultime regole in Figura 6.17.

6.2.2 La compilazione condizionale delle espressioni booleane

Abbiamo descritto come un'espressione viene tradotta in del bytecode Kitten che ne calcola il valore e lo lascia in cima allo stack degli operandi. Tale codice è adeguato se quello a cui siamo interessati è il valore dell'espressione. Per esempio, di un parametro passato a un metodo abbiamo bisogno del valore, così come del lato destro di un assegnamento. Ci sono casi però in cui quello che ci interessa è di instradare l'esecuzione di un programma in due direzioni diverse sulla base del valore di un'espressione booleana. Per esempio, nel comando `if` (*exp*) `then` com_1 `else` com_2 siamo interessati ad eseguire com_1 se il valore di *exp* è *true* e ad eseguire com_2 se tale valore è invece *false*. Occorre quindi definire un altro modo di generare il bytecode per le espressioni, alternativo a quello della Figura 6.17 e che chiameremo *compilazione condizionale* delle espressioni. Va comunque detto che ricicleremo in larghissima misura le

definizioni in tale figura. Va ricordato inoltre che la compilazione condizionale ha senso solo per le espressioni che hanno tipo `boolean`, dal momento che l'analisi semantica ci garantisce che esse sono le uniche che possono essere usate nei test dei condizionali e dei cicli (Figura 5.11).

Definiamo quindi una funzione

$$\gamma^{test}[\![\_]\!] : \texttt{absyn.Expression} \to \texttt{translate.CodeBlock} \mapsto \texttt{translate.CodeBlock} \mapsto \texttt{translate.CodeBlock}$$

che compila un'espressione in maniera condizionale. In particolare, $\gamma^{test}[\![exp]\!](\beta_{true})(\beta_{false})$ è la compilazione condizionale dell'espressione *exp*: se l'espressione contiene *true* l'esecuzione viene instradata verso la continuazione β_{true}; altrimenti verso la continuazione β_{false}. La sua definizione sfrutta quella in Figura 6.17:

$$\gamma^{test}[\![exp]\!](\beta_{true})(\beta_{false}) = \gamma[\![exp]\!]\left(\boxed{\texttt{nop}} \left\langle \begin{array}{l} \boxed{\texttt{if\_true}} \to \beta_{true} \\ \boxed{\texttt{if\_false}} \to \beta_{false} \end{array} \right.\right) \tag{6.2}$$

Per esempio, supponendo che la variabile `i` sia allocata nella variabile locale numero 1 e che abbia tipo `int`, allora la compilazione condizionale di `i < 5` è

$$\boxed{\begin{array}{l}\texttt{load 1 of type int} \\ \texttt{const 5} \\ \texttt{lt int}\end{array}} \to \boxed{\texttt{nop}} \left\langle \begin{array}{l} \boxed{\texttt{if\_true}} \to \beta_{true} \\ \boxed{\texttt{if\_false}} \to \beta_{false} \end{array} \right.$$

La funzione $\gamma^{test}[\![\_]\!]$ è implementata aggiungendo ad `absyn.Expression` il metodo:

```
public CodeBlock translateAsTest(CodeBlock yes, CodeBlock no) {
  return translate(new CodeBlock(new IF_TRUE(),yes,no));
}
```

Il costruttore utilizzato per questo `CodeBlock` costruisce un blocco con codice `nop` e legato alle continuazioni `yes` e `no` tramite, rispettivamente, il bytecode condizionale `if_true` e il suo opposto.

La definizione di $\gamma^{test}[\![\_]\!]$ che abbiamo appena visto funziona per qualsiasi espressione condizionale. Genera però del codice particolarmente ridondante. Per esempio, la compilazione condizionale di `i < 5` che abbiamo ottenuto sopra è molto meno ottimizzata di quella in Figura 6.3, che non usa né l'istruzione `nop` né la `lt int` e usa invece i bytecode condizionali `if_cmplt int` ed `if_cmpge int` al posto di `if_true` ed `if_false`. Il problema della `nop` non deve preoccuparci: una volta generato il bytecode per una classe Kitten, elimineremo tutte le `nop` dal codice. Per usare invece dei bytecode condizionali specializzati, possiamo aggiungere delle definizioni specifiche per la funzione $\gamma^{test}[\![\_]\!]$, che ridefiniscono la precedente definizione generale su dei casi particolari molto frequenti. Per esempio definiamo

$$\gamma^{test}[\![\texttt{LessThan}(left, right)]\!](\beta_{true})(\beta_{false}) = \gamma^{\ell}[\![left]\!]\left(\gamma^{\ell}[\![right]\!]\left(\boxed{\texttt{nop}} \left\langle \begin{array}{l} \boxed{\texttt{if\_cmplt}} \to \beta_{true} \\ \boxed{\texttt{if\_cmpge}} \to \beta_{false} \end{array} \right.\right)\right)$$

dove ℓ è il minimo sovratipo comune del tipo statico di *left* e *right*. Dal punto di vista implementativo, queste ridefinizioni diventano delle ridefinizioni del metodo `translateAsTest()` in alcune sottoclassi di `absyn.Expression`.

lvalue	*bytecode*
`Variable`(*name*)	$\gamma^{\tau}[\![rvalue]\!]\left(\boxed{\texttt{store}\ num\ \texttt{of type}\ \tau} \rightarrow \beta\right)$
`FieldAccess`(*receiver*, *name*)	$\gamma[\![receiver]\!]\left(\gamma^{\tau}[\![rvalue]\!]\left(\boxed{\texttt{putfield}\ field} \rightarrow \beta\right)\right)$
`ArrayAccess`(*array*, *index*)	$\gamma[\![array]\!]\left(\gamma[\![index]\!]\left(\gamma^{\tau}[\![rvalue]\!]\left(\boxed{\begin{array}{c}\texttt{arraystore into}\\ \texttt{array of}\ \tau\end{array}} \rightarrow \beta\right)\right)\right)$

Figura 6.18: La compilazione passiva di un leftvalue di tipo statico τ.

6.2.3 La compilazione passiva dei leftvalue

I leftvalue sono un caso particolare di espressioni (Sezione 3.2.3). Abbiamo quindi già specificato per essi una modalità di compilazione che lascia il loro valore in cima allo stack degli operandi (Sezione 6.2.1 e Figura 6.17), che usiamo quando del leftvalue ci interessa il valore, come per `a[6]` in `v := a[6]`, e un'altra modalità che instrada l'esecuzione verso due direzioni diverse sulla base del valore booleano che essi contengono (Sezione 6.2.2 ed Equazione 6.2), che usiamo quando il leftvalue è usato come test booleano, per esempio per `a[8 + v]` in `if (a[8 + v]) then...else...` A differenza delle altre espressioni, i leftvalue possono però essere usati anche alla sinistra di un assegnamento, come `v` in `v := b + c` oppure `a[5]` in `a[5] := b * c`. In questi casi non siamo interessati al valore del leftvalue, né a instradare l'esecuzione su due continuazioni diverse sulla base del valore booleano del leftvalue. Vogliamo invece *modificare* il valore del leftvalue. Conseguentemente, dobbiamo definire una terza modalità di compilazione per i leftvalue, che chiameremo *passiva* poiché il leftvalue subisce un assegnamento.

Si consideri un assegnamento del tipo *lvalue* := *rvalue*. Vogliamo generare il codice che effettua l'assegnamento e poi continua con una continuazione β. Sia τ il tipo statico di *lvalue*. Il bytecode che generiamo è mostrato in Figura 6.18. Essa mostra che la compilazione passiva di un leftvalue è sempre della forma

$$\gamma^{before}[\![lvalue]\!](\gamma^{\tau}[\![rvalue]\!](\gamma^{after}[\![lvalue]\!](\beta)))$$

dove $\gamma^{before}[\![\_]\!], \gamma^{after}[\![\_]\!]$: `absyn.Lvalue` $\mapsto$ `absyn.CodeBlock` $\mapsto$ `absyn.CodeBlock` sono due funzioni che aggiungono del codice, rispettivamente, prima e dopo la compilazione di *rvalue*. Si noti che quest'ultimo è compilato rispetto al tipo τ di *lvalue*, in modo da effettuare una promozione di tipo quando *rvalue* ha tipo `int` e lo si sta assegnando a un *lvalue* di tipo `float` (Sezione 6.2). Le funzioni γ^{before} e γ^{after} sono implementate aggiungendo a `absyn/Lvalue.java` i due metodi

```
public abstract CodeBlock translateBeforeAssignment(CodeBlock continuation);

public abstract CodeBlock translateAfterAssignment(CodeBlock continuation);
```

che vengono istanziati nelle sottoclassi in modo da rispettare la Figura 6.18. Per esempio, dentro `absyn/Variable.java` sono ridefiniti come

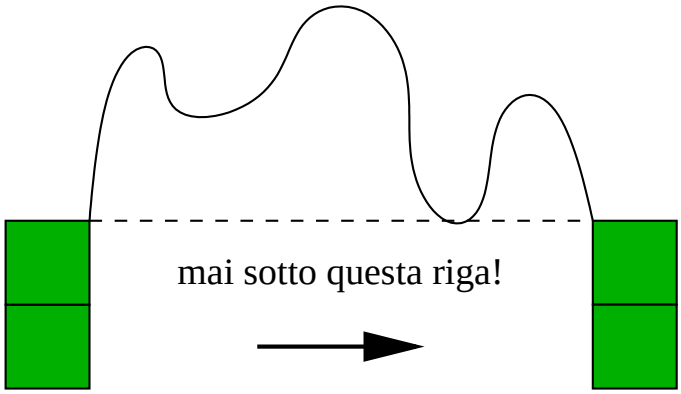


Figura 6.19: L'esecuzione del bytecode Kitten generato per un comando non deve modificare lo stack degli operandi iniziale.

```
public CodeBlock translateBeforeAssignment(CodeBlock continuation) {
  return continuation;
}

public CodeBlock translateAfterAssignment(CodeBlock continuation) {
  return new STORE(getVarNum(),getStaticType()).followedBy(continuation);
}
```

Dentro `absyn/ArrayAccess.java` sono ridefiniti come

```
public CodeBlock translateBeforeAssignment(CodeBlock continuation) {
  return array.translate(index.translate(continuation));
}

public CodeBlock translateAfterAssignment(CodeBlock continuation) {
  return new ARRAYSTORE(getStaticType()).followedBy(continuation);
}
```

Questi due metodi sono usati per compilare il comando di assegnamento, come vedremo nella prossima sezione.

6.3 La generazione del bytecode Kitten per i comandi

La generazione del bytecode per un comando Kitten è formalizzata tramite una funzione $\gamma[\![\_]\!]$: `absyn.Command` $\mapsto$ `translate.CodeBlock` $\mapsto$ `translate.CodeBlock`. Dato un comando *com* e una continuazione β, il codice $\gamma[\![com]\!](\beta)$ dovrà essere del bytecode Kitten che esegue il comando *com* e poi continua eseguendo la continuazione β. Il codice generato per eseguire i comandi deve essere tale da lasciare intatti i valori iniziali sullo stack degli operandi. Si tratta

$$\gamma[\![\_]\!] : \texttt{absyn.Command} \mapsto (\texttt{translate.CodeBlock} \mapsto \texttt{translate.CodeBlock})$$

$$\gamma[\![\texttt{Skip}()]\!](\beta) = \beta \qquad \gamma[\![\texttt{LocalScope}(\mathit{body})]\!](\beta) = \gamma[\![\mathit{body}]\!](\beta)$$

$$\gamma[\![\texttt{Return}(\mathit{returned})]\!](\beta) = \begin{cases} \boxed{\texttt{return void}} & \text{se } \mathit{returned} = \texttt{null} \\ \gamma[\![\mathit{returned}]\!]\left(\boxed{\texttt{return } \tau}\right) & \text{se } \mathit{returned} \neq \texttt{null} \text{ e ha tipo statico } \tau \end{cases}$$

$$\gamma[\![\texttt{IfThenElse}(\mathit{condition}, \mathit{then}, \mathit{else})]\!](\beta) = \gamma^{\mathit{test}}[\![\mathit{condition}]\!](\gamma[\![\mathit{then}]\!](\beta))(\gamma[\![\mathit{else}]\!](\beta))$$

$$\gamma[\![\texttt{LocalDeclaration}(\mathit{type}, \mathit{name}, \mathit{initialiser})]\!](\beta) = \gamma^{\tau}[\![\mathit{initialiser}]\!]\left(\boxed{\texttt{store } \mathit{num} \texttt{ of type } \tau} \to \beta\right)$$

dove τ è il tipo semantico di *type* e *num* è il numero progressivo della variabile *name*

$$\gamma[\![\texttt{MethodCallCommand}(\mathit{receiver}, \mathit{name}, \mathit{actuals})]\!](\beta) = \begin{cases} \gamma[\![\mathit{receiver}]\!]\left(\gamma^{\vec{t}}[\![\mathit{actuals}]\!]\left(\boxed{\texttt{virtualcall } \mathit{method}} \to \beta\right)\right) \\ \quad \text{se } t' = \texttt{void} \\ \gamma[\![\mathit{receiver}]\!]\left(\gamma^{\vec{t}}[\![\mathit{actuals}]\!]\left(\boxed{\begin{array}{l}\texttt{virtualcall } \mathit{method} \\ \texttt{pop } t'\end{array}} \to \beta\right)\right) \\ \quad \text{altrimenti} \end{cases}$$

dove $\mathit{method} = \kappa.\mathtt{m}(\vec{t}) : t'$ è il metodo identificato dall'analisi semantica (Figura 5.11)

$$\gamma[\![\texttt{Assignment}(\mathit{lvalue}, \mathit{rvalue})]\!](\beta) = \gamma^{\mathit{before}}[\![\mathit{lvalue}]\!]\left(\gamma^{\tau}[\![\mathit{rvalue}]\!](\gamma^{\mathit{after}}[\![\mathit{lvalue}]\!](\beta))\right)$$

dove τ è il tipo statico di *lvalue*

$$\gamma[\![\texttt{While}(\mathit{condition}, \mathit{body})]\!](\beta) = \underbrace{\boxed{\texttt{nop}}}_{\mathit{pivot}} \to \gamma^{\mathit{test}}[\![\mathit{condition}]\!](\gamma[\![\mathit{body}]\!](\mathit{pivot}))(\beta)$$

$$\gamma[\![\texttt{For}(\mathit{initialiser}, \mathit{condition}, \mathit{update}, \mathit{body})]\!](\beta) = \gamma[\![\mathit{initialiser}]\!]\left(\underbrace{\boxed{\texttt{nop}}}_{\mathit{pivot}} \to \gamma^{\mathit{test}}[\![\mathit{condition}]\!]\left(\gamma[\![\mathit{body}]\!]\left(\gamma[\![\mathit{update}]\!](\mathit{pivot})\right)\right)(\beta)\right)$$

Figura 6.20: La funzione $\gamma[\![\_]\!]$ che genera il bytecode Kitten che esegue i comandi.

esattamente dello stesso vincolo imposto al codice generato per le espressioni nella Sezione 6.2. In tal caso si chiedeva però anche che il valore dell'espressione fosse aggiunto in cima allo stack degli operandi. Dal momento che i comandi non calcolano alcun valore, non esiste per essi tale secondo vincolo. Il comportamento del bytecode generato per i comandi sarà quindi come mostrato in Figura 6.19.

La Figura 6.20 mostra il codice generato per i comandi Kitten. Commentiamo tali regole di compilazione.

`Skip`()**.** Questo comando non genera alcun bytecode e quindi la sua compilazione restituisce la continuazione β.

`LocalScope`(*body*)**.** L'esecuzione di uno scope locale consiste nell'esecuzione del suo corpo. Conseguentemente, la sua compilazione è, ricorsivamente, la compilazione del suo corpo.

`Return`(*returned*)**.** L'istruzione di ritorno da metodo viene tradotta in un bytecode `return` per il tipo del valore ritornato, se esiste. In tal caso occorre prima compilare l'espressione il cui valore va ritornato. Si noti che la continuazione β è scartata poiché l'esecuzione di un metodo termina col ritorno al chiamante.

`IfThenElse`(*condition*, *then*, *else*)**.** La compilazione del condizionale comincia con la compilazione come test della sua guardia (Sezione 6.2.2). Le due continuazioni della guardia sono, rispettivamente, la compilazione del ramo `then` e del ramo `else` del condizionale, seguite dalla continuazione β del condizionale.

`LocalDeclaration`(*type*, *name*, *initialiser*)**.** La compilazione della dichiarazione di una variabile locale, con inizializzazione, è del codice che valuta l'inizializzatore e ne lascia il valore in cima allo stack, da cui è poi rimosso e scritto dentro alla variabile tramite un bytecode `store`. Si noti che il numero *num* della variabile è stato assegnato al momento dell'analisi semantica.

`MethodCallCommand`(*receiver*, *name*, *actuals*)**.** La compilazione del comando di invocazione di metodo è quasi identica a quella che abbiamo visto per l'espressione di invocazione di metodo (Figura 6.17). La differenza è che qui è possibile invocare anche un metodo che ritorna `void`. Inoltre, dal momento che non dobbiamo modificare lo stack degli operandi (Figura 6.19), rimuoviamo il valore di ritorno di un metodo non `void` tramite un bytecode `pop`.

`Assignment`(*lvalue*, *rvalue*)**.** La compilazione di un assegnamento di *rvalue* ad *lvalue* è ottenuta come in Figura 6.18.

`While`(*condition*, *body*)**.** Il codice generato per un ciclo `while` è la compilazione condizionale della sua guardia (Sezione 6.2.2), usando come due continuazioni quella stessa del `while`, per il caso in cui la guardia è falsa, e la compilazione del corpo per il caso in cui la guardia è vera. Si noti che la continuazione fornita alla compilazione del corpo è un blocco *pivot* che continua con la compilazione condizionale della guardia stessa, in modo che dopo l'esecuzione del corpo del `while` si passi a valutare di nuovo la guardia del ciclo.

`For`(*initialiser*, *condition*, *update*, *body*)**.** Il codice generato per un ciclo `for` comincia con il codice che esegue il comando di inizializzazione, seguito da un blocco *pivot* legato alla compilazione condizionale della guardia del `for` (Sezione 6.2.2). Le due continuazioni passate a tale compilazione condizionale sono la continuazione β del `for`, per il caso in cui la guardia è falsa, e la compilazione dell'*update* e del corpo del ciclo per il caso in cui la guardia è vera. Si noti che la continuazione usata per la compilazione del corpo è il *pivot*, in modo che dopo l'esecuzione del corpo del `for` si torni a valutare la guardia del ciclo.

L'implementazione della generazione del codice per i comandi è ottenuta aggiungendo i seguenti metodi ad `absyn/Command.java`:

```
public final CodeBlock translate(CodeBlock continuation) {
  if (next != null) continuation = next.translate(continuation);
  return translate$0(continuation);
}

protected abstract CodeBlock translate$0(CodeBlock continuation);
```

Il primo si occupa del lavoro comune a tutti i comandi, che consiste nel compilare il comando che potrebbe seguire ottenendo la continuazione da passare al metodo `translate$0()`. Quest'ultimo si occupa del lavoro specifico ad ogni comando. Esso implementa le regole in Figura 6.20.

Vediamo alcuni esempi di definizione di `translate$0()` in alcune delle sottoclassi della classe `absyn/Command.java`. Dentro `absyn/LocalScope.java` definiamo

```
protected CodeBlock translate$0(CodeBlock continuation) {
  return body.translate(continuation);
}
```

consistentemente con la Figura 6.20. In `absyn/IfThenElse.java` definiamo

```
protected CodeBlock translate$0(CodeBlock continuation) {
  return condition.translateAsTest
    (then.translate(continuation),else.translate(continuation));
}
```

ancora una volta questo rispecchia la formalizzazione in Figura 6.20.

L'implementazione delle regole per il `while` e il `for` richiedono di creare *prima* il *pivot* in modo da poterlo passare come continuazione, rispettivamente, alla compilazione del *body* o dell'*update* del ciclo. Alla fine si lega il blocco *pivot* con il suo successore, chiudendo il ciclo. Ecco per esempio il generatore di codice inserito dentro `absyn/For.java`:

```
protected CodeBlock translate$0(CodeBlock continuation) {
  CodeBlock pivot = new CodeBlock();
  CodeBlock test = condition.translateAsTest
    (body.translate(update.translate(pivot)),continuation);
  pivot.linkTo(test);
  return initialisation.translate(test);
}
```

Si noti che il blocco *pivot* va creato prima di usarlo come continuazione per la compilazione di `update`, nel caso del comando `for`. Sarebbe sbagliato dichiarare la variabile `pivot` e creare il blocco *pivot* subito prima della chiamata a `linkTo()`: l'`update` si troverebbe con una continuazione pari a `null`!

La generazione del bytecode Kitten per un metodo o costruttore è semplicemente la generazione del bytecode Kitten per il loro corpo, che essendo un comando segue le regole in Figura 6.20. Come continuazione di tale compilazione si usa il blocco

$$\overline{\beta} = \boxed{\texttt{return void}}\,.$$

In questo modo abbiamo la garanzia che, nel bytecode che viene generato, ogni percorso di esecuzione all'interno di un metodo che ritorna `void` o all'interno di un costruttore termina sempre con un'istruzione `return void`, anche nei casi in cui il comando `return` è stato lasciato

sottointeso dal programmatore. Si noti che nel caso in cui fossimo dentro un metodo che non ritorna `void` allora tale continuazione verrebbe sistematicamente scartata dalla regola per il comando Kitten `return` in Figura 6.20, dal momento abbiamo la garanzia che, in tal caso, ogni percorso di esecuzione all'interno del metodo termina già con un comando `return` esplicito (Sezione 4.3).

Esercizio 25. Si parta dalla sintassi astratta dell'espressione condizionale definita nell'Esercizio 22 e si scriva la sua funzione γ di compilazione, implementandola poi in Java.

Esercizio 26. Si definisca la sintassi astratta di un comando `do...while` e si dia quindi la sua funzione γ di compilazione, implementandola poi in Java.

Esercizio 27. Si parta dalla sintassi astratta del comando `switch` definito nell'Esercizio 23 e si definisca la sua funzione γ di compilazione, implementandola poi in Java.

Esercizio 28. Quali problemi vedete per definire la compilazione dei comandi `break` e `continue` dell'Esercizio 24? Come pensate di poter modificare lo schema di compilazione per continuazioni in modo da poter compilare tali due comandi?

Bibliografia

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles Techniques and Tools*. Addison Wesley Publishing Company, 1986.

[2] A. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, 2002.

[3] A. Goldberg and D. Robson. *Smalltalk 80: The Language*. Addison-Wesley Professional, 1989.

[4] J. Gosling, B. Joy, Guy Steel, and G. Bracha. *The Java^{TM} Language Specification*. Addison-Wesley, third edition, 2005.

[5] S. C. Johnson. Yacc - Yet Another Compiler Compiler. Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[6] M. E. Lesk. Lex - A Lexical Analyzer Generator. Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.