Capitolo 1

Contesto e fasi di un compilatore

Come abbiamo visto nella parte dedicata alla definizione di compilazione e traduzione, una implementazione compilativa di un linguaggio di programmazione è realizzata tramite un programma (il compilatore) che, preso un altro programma scritto nel linguaggio *sorgente*, restituisce un programma *funzionalmente equivalente* a quello dato, ma scritto in un linguaggio *target*, che è il linguaggio della macchina intermedia scelta per l'implementazione. Il compilatore in sé può essere scritto in qualsiasi linguaggio e girare su qualsiasi macchina astratta.

La conoscenza di tecniche efficaci per la scrittura di un programma compilatore è ad oggi molto vasta e strutturata. Non era così per i primi programmatori che alla fine degli anni '50 si accingevano a scriverne uno. Si pensi ad esempio che la scrittura del primo compilatore Fortran richiese 18 anni di lavoro da parte di uno staff. Sin da allora sono state individuate molte tecniche sistematiche per la scrittura del codice delle fasi più importanti di un compilatore, oltre ad essere sorti linguaggi di programmazione adatti (ad esempio il linguaggio C) per l'implementazione, ambienti di programmazione e tool software (generatori automatici di codice a partire dalle specifiche di varie parti del linguaggio sorgente).

Il processo di compilazione è concettualmente diviso in due parti: l'analisi e la sintesi. La parte di analisi si occupa di dare una struttura al codice sorgente e di creare una sua rappresentazione intermedia. La parte di sintesi genera il codice nel linguaggio target a partire dalla rappresentazione intermedia.

Durante la fase di analisi le operazioni indicate nel programma sorgente vengono riconosciute e raggruppate in una struttura ad albero. Spesso viene usato il cosiddetto *albero sintattico* (syntax tree), cioè un albero in cui

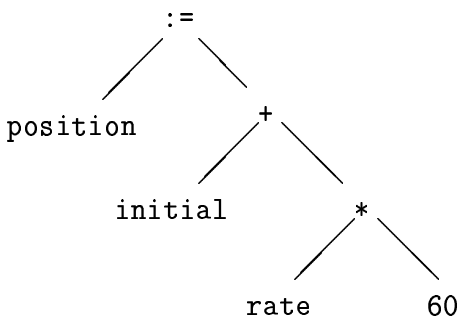

Figura 1.1: Albero sintattico per `position := initial + rate * 60`

ogni nodo interno rappresenta un operatore i cui operandi sono reperibili nei figli. Ad esempio, in Figura 1.1 è raffigurato un albero sintattico per l'assegnamento Pascal `position := initial + rate * 60`

1.1 Contesto di un compilatore

Per poter generare un programma target eseguibile sulla macchina ospite, altri tipi di programmi vengono in aiuto del compilatore. Ciò per consentirgli di concentrarsi solo sulla traduzione.

In molti casi succede che il programma sorgente sia diviso in moduli che si trovano su file diversi. Il compito di recuperare tutti i file e di creare un file unico sorgente è spesso affidato ad un *preprocessore*. Esso, in genere, si occupa anche di espandere le macro che possono essere presenti nel programma sorgente originale. Un esempio è il preprocessore per il compilatore del linguaggio C. Esso si occupa di rimpiazzare le direttive `#include` con il testo contenuto nei file specificati. Inoltre rimpiazza nel codice le definizioni di macro definite dalle direttive `#define`.

L'output della traduzione spesso necessita di altre trasformazioni prima di poter essere effettivamente eseguito sulla macchina ospite. Ad esempio non è raro trovare dei compilatori che generano come codice target un programma scritto nel codice assembly della macchina ospite. Questo codice deve essere assemblato tramite un *assemblatore* per diventare codice macchina rilocabile.

Inoltre abbiamo visto che la macchina intermedia utilizzata nell'approccio compilativo viene simulata sulla macchina ospite e spesso questa simulazione viene implementata tramite un supporto a tempo di esecuzione. Riprendendo l'esempio del linguaggio C, nei sistemi unix il supporto a tempo di esecuzione per il C è formato da librerie di funzioni rilocabili (che esistono in unica copia in directory apposite del file system) che vengono caricate solo quando il programma va in esecuzione. Il *caricatore* si preoccupa di reperire questo

codice rilocabile condiviso e di rilocare tutto il codice così ottenuto. Il risultato è un codice macchina assoluto che può essere eseguito sulla macchina ospite. In Figura 1.2 è raffigurato questo esempio. La rilocazione del codice consiste nel convertire gli indirizzi relativi all'interno dello spazio logico del programma in indirizzi assoluti (a seconda dell'effettiva zona della memoria in cui il programma viene caricato). In molti casi, a seconda della macchina hardware e del sistema operativo, la rilocazione viene fatta a livello hardware o firmware.

1.2 Analisi

In questa sezione introdurremo in maniera non dettagliata le fasi che compongono la parte di analisi. Per ognuna di queste fasi, approfondiremo le tecniche e gli algoritmi che le sono propri nei capitoli successivi.

L'analisi viene divisa in tre fasi:

1. Analisi lineare: lo stream di caratteri che costituiscono il programma sorgente è letto da sinistra a destra e raggruppato in *token* che sono sequenze di caratteri che hanno un significato comune.

2. Analisi gerarchica: i token vengono raggruppati gerarchicamente in strutture annidate (alberi) che specificano la struttura delle frasi del linguaggio.

3. Analisi semantica: attraverso vari controlli si cerca di appurare se le varie parti del programma sono consistenti una con l'altra, a seconda del loro significato.

1.2.1 Analisi lessicale

In un compilatore, l'analisi lineare è chiamata analisi lessicale o *scanning*. Durante questa analisi una linea di caratteri come la seguente

```
position := initial + rate * 60
```

viene, in genere, raggruppata nei seguenti token:

1. L'identificatore `position`

2. Il simbolo di assegnamento: `:=`

3. L'identificatore `initial`

4. Il segno dell'addizione

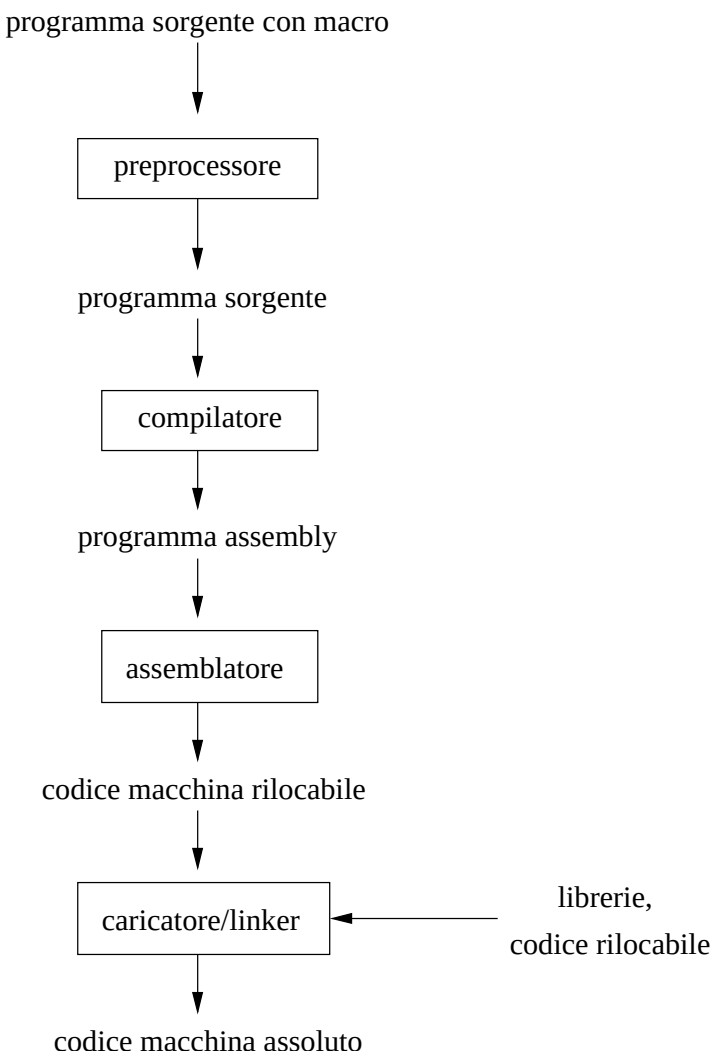

Figura 1.2: Esempio di un contesto di un compilatore.

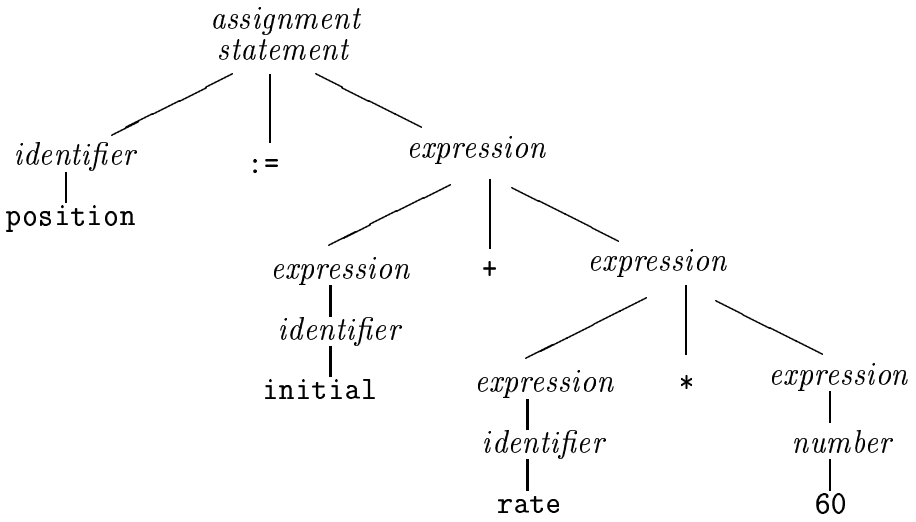

Figura 1.3: Parse tree per `position := initial + rate * 60`

5. L'identificatore `rate`
6. Il segno della moltiplicazione
7. Il numero `60`

I caratteri di spazio (ma anche tabulazioni, commenti e caratteri che indicano la fine di una linea) che separano i token vengono, normalmente, eliminati durante l'analisi lessicale.

1.2.2 Analisi sintattica

L'analisi gerarchica viene chiamata *parsing* o analisi sintattica. Il suo compito è di raggruppare i token in frasi grammaticali che poi saranno usate per sintetizzare l'output. Generalmente le frasi grammaticali del programma sorgente sono rappresentate con un albero di derivazione (o *parse tree*) di una grammatica libera dal contesto. Vedremo questo formalismo più in dettaglio nel Capitolo 3. In Figura 1.3 è raffigurato un parse tree per i token dell'esempio della sezione precedente.

La divisione fra analisi lessicale e sintattica è una scelta del progettista del compilatore e non ci sono direttive prestabilite. In genere la scelta ricade sulla soluzione che semplifica di più l'analisi sintattica. Un fattore determinante per la divisione è l'intrinseca ricorsività di un costrutto. I costrutti lessicali non richiedono la ricorsione, mentre i costrutti sintattici spesso si. E infatti le grammatiche libere dal contesto sono una formalizzazione di regole ricorsive che possono essere usate per guidare l'analisi sintattica.

Facciamo un esempio: la ricorsione non è necessaria per riconoscere gli identificatori che, tipicamente, sono stringhe che cominciano con una lettera e continuano con cifre o lettere. Per riconoscere un identificatore, quindi, basta vedere una lettera e continuare a leggere l'input fino a che non si incontra un carattere che non è né una cifra né una lettera. Quando questo accade basta raggruppare tutti i caratteri letti, meno l'ultimo, e considerarli come un token identificatore. Questi caratteri raggruppati vengono poi inseriti nella *tabella dei simboli* (che è una struttura dati molto importante usata in tutte le fasi della compilazione) e rimossi dall'input in modo che si possa continuare la ricerca del token successivo.

D'altra parte, questo tipo di scansione lineare non è abbastanza potente per poter analizzare espressioni o *statement*. Ad esempio non si possono riconoscere stringhe in cui le parentesi aperte e chiuse sono in egual numero, oppure non si possono associare nella maniera giusta i `begin` e gli `end` dei costrutti senza dare una struttura gerarchica o annidata all'input.

Il parse tree in Figura 1.3 descrive la struttura sintattica dell'input. Una rappresentazione più concisa e per questo più usata come rappresentazione interna è quella dell'albero sintattico, di cui abbiamo un esempio in Figura 1.1 (questo albero è il corrispondente più conciso dell'albero di derivazione di Figura 1.3). Approfondiremo l'uso di queste strutture dati nel Capitolo 3 e anche nel Capitolo 4, dove il compilatore usa la struttura gerarchica dell'input per generare il suo output.

1.2.3 Analisi semantica

La fase di analisi semantica controlla se ci sono errori semantici nel programma sorgente e acquisisce l'informazione sui tipi che verrà usata nella fase successiva di generazione del codice. La rappresentazione gerarchica generata dall'analisi sintattica viene usata per identificare gli operatori e gli operandi delle espressioni e degli statement.

Una componente importante di questa fase è il *type checking* in cui il compilatore controlla che ogni operatore viene applicato ad operandi consentiti dalla specifica del linguaggio. Un esempio: la definizione di molti linguaggi di programmazione richiede che sia riportato un errore se un numero reale è utilizzato per indicizzare un array.

Tuttavia alcuni "abusi" sull'uso degli operandi potrebbero essere permessi, come ad esempio l'applicazione di un operatore aritmetico binario ad un numero intero e ad un reale. In questo caso il compilatore può dover convertire l'intero in reale per ottenere un risultato coerente. Supponiamo, infatti, che il type checking ha rivelato che l'identificatore `rate` nella nostra stringa di esempio abbia tipo `real` e che il numero `60` sia considerato di per se stesso

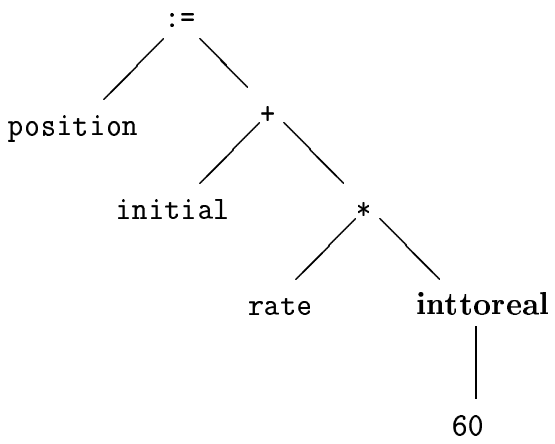

Figura 1.4: Albero sintattico dopo l'analisi semantica.

come un intero. Generalmente in questi casi il compilatore deve convertire il numero intero in numero reale poiché la rappresentazione interna degli interi e dei reali in genere è diversa. Questa operazione viene segnalata inserendo un ulteriore nodo interno nell'albero sintattico e sarà effettuata nella fase di traduzione (l'albero sintattico diventa come in Figura 1.4).

1.3 Le fasi di un compilatore

Concettualmente un compilatore opera in diverse fasi ognuna delle quali trasforma il programma sorgente da una rappresentazione ad un'altra. Una tipica decomposizione in fasi è mostrata in Figura 1.5. Nella reale implementazione, poi, le fasi vengono spesso accorpate e le rappresentazioni intermedie che ci sono tra le fasi raggruppate non vengono costruite esplicitamente.

Dalla figura si notano due componenti, il gestore della tabella dei simboli e il gestore degli errori, che interagiscono con tutte le fasi. Le chiameremo informalmente "fasi", ma non lo sono propriamente.

1.3.1 Gestione della tabella dei simboli

Una funzione essenziale di un compilatore è quella di memorizzare gli identificatori che vengono usati nel programma sorgente insieme con i relativi valori di diversi attributi. Questi attributi possono essere, ad esempio, lo spazio in byte allocato per l'identificatore, il suo tipo, il suo *scope* (la parte di programma sorgente in cui l'identificatore ha significato). Ancora, se l'identificatore è il nome di una procedura, altri tipi di attributi sono il numero e i tipi di

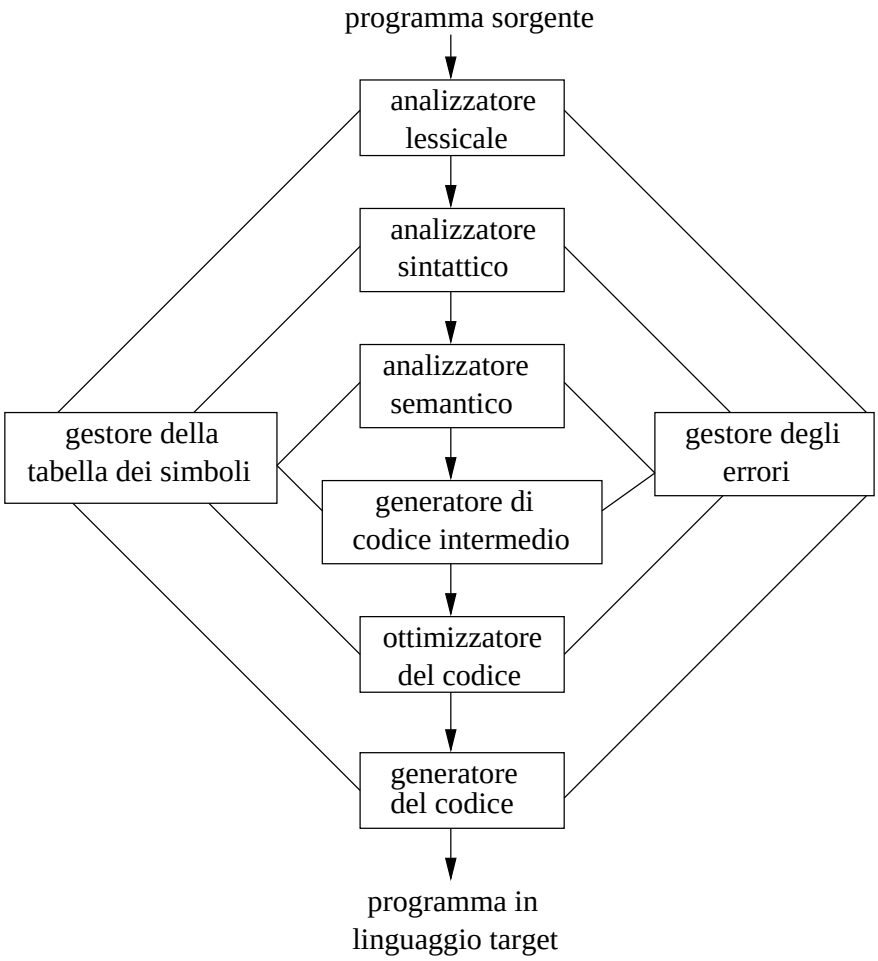

Figura 1.5: Fasi di un compilatore.

parametri che essa richiede, la modalità di passaggio di questi parametri (per valore, per riferimento, per nome), il tipo del valore ritornato, se c'è.

La tabella dei simboli è una struttura dati che contiene un record per ogni identificatore e in cui i vari campi del record sono gli attributi. Questa struttura dati deve consentire di trovare efficientemente il record di un identificatore che stiamo cercando e di leggere/scrivere nuovi valori in maniera efficiente.

Quando un nuovo identificatore viene trovato nel codice sorgente durante la fase di analisi lessicale esso viene inserito nella tabella dei simboli dall'analizzatore. Tuttavia i valori degli attributi non possono essere trovati tutti durante questa fase. Ad esempio, nella dichiarazione Pascal:

```
var position, initial, rate :  real;
```

il tipo `real` non è ancora conosciuto quando gli identificatori `position, initial, rate` sono letti dall'analizzatore lessicale.

Le fasi rimanenti inseriscono informazioni nella tabella dei simboli e/o le usano in vari modi. Ad esempio, durante le fasi di analisi semantica e di generazione del codice intermedio c'è bisogno di sapere il tipo degli identificatori per controllare che siano usati propriamente e per generare le corrette operazioni su di essi. La fase di generazione del codice inoltre aggiunge alla tabella dei simboli anche informazioni dettagliate sullo spazio di memoria allocato per ogni identificatore.

1.3.2 Rilevazione e presentazione degli errori

Una parte importante del processo di compilazione, oltre alla traduzione vera e propria, è il riconoscimento e la segnalazione di errori nel programma sorgente. Ogni fase può incontrare errori e, dopo averne rilevato uno, essa deve in qualche modo trattarlo per fare in modo che la compilazione continui e altri eventuali errori possano essere rilevati. Questo perchè un compilatore che si fermi una volta che ha trovato il primo errore non è così di aiuto come potrebbe essere.

Le fasi di analisi sintattica e semantica sono quelle che devono fronteggiare la frazione maggiore di errori che un compilatore è in grado di rilevare. L'analizzatore lessicale può rilevare un errore quando i caratteri rimanenti dell'input non formano nessun token del linguaggio. Quando lo stream di token vìola le regole strutturali del linguaggio (la sintassi), il parser rileva un errore. Durante l'analisi semantica possono essere riconosciuti i costrutti che, pur essendo sintatticamente corretti, implicano operazioni non consentite fra gli operandi in gioco. Si pensi ad esempio all'addizione fra una variabile di

un tipo primitivo ed un array[1]. Quando parleremo dettagliatamente di ogni fase faremo anche degli accenni a come poter gestire gli errori che tipicamente possono sorgere.

1.3.3 Le fasi di analisi

Abbiamo già parlato delle prime tre fasi. In questa sezione però precisiamo alcune cose sulla rappresentazione interna che esse generano.

La parte di analisi lessicale, abbiamo visto, raggruppa i caratteri del programma sorgente in token, che sono sequenze di caratteri che hanno un significato comune (una categoria lessicale), come ad esempio gli identificatori, le parole chiave (`if, then, while`, ecc.) i segni di punteggiatura, gli operatori multicarattere (ad esempio `:=`). La sequenza di caratteri che forma un certo token è chiamata *lexeme* del token in questione. Oltre all'informazione sul token stesso, alcune categorie lessicali richiedono che un'altra informazione sia ad esso associata: il "valore lessicale".

Ad esempio, quando viene trovato un identificatore come `rate`, l'analizzatore lessicale genera un token, chiamiamolo **id**, e inserisce il lexeme `rate` nella tabella dei simboli (se non c'è già). Il valore lessicale associato a questa occorrenza del token **id** sarà un puntatore all'entrata della tabella dei simboli che contiene il lexeme (e, abbiamo visto, anche altri attributi).

In questa sezione useremo $\mathbf{id}_1$, $\mathbf{id}_2$ ed $\mathbf{id}_3$ in luogo di `position`, `initial` e `rate` proprio per enfatizzare il fatto che la rappresentazione interna di un identificatore è diversa dal lexeme. Quindi la nostra stringa di esempio, dopo l'analisi lessicale, ha la seguente rappresentazione:

$\mathbf{id}_1$ `:=` $\mathbf{id}_2$ `+` $\mathbf{id}_3$ `*` `60`

Avremmo dovuto definire anche la rappresentazione degli altri token, ma posticipiamo questo discorso al Capitolo 2. La rappresentazione del codice che viene fuori dalle due fasi successive cambia in accordo a questa rappresentazione dei token. La Figura 1.6 illustra questa nuova situazione e tutta la traduzione della nostra stringa di esempio.

1.3.4 Generazione del codice intermedio

Dopo l'analisi sintattica e semantica alcuni compilatori generano una rappresentazione intermedia esplicita del programma sorgente. Possiamo pensare a questa rappresentazione intermedia come ad un programma scritto nel linguaggio di una macchina astratta che è poco distante dalla macchina

[1] Nulla toglie ovviamente che il linguaggio possa prevedere questo tipo di operazione, ma in ogni caso essa non deve essere trattata come una normale addizione, bensì dovrà essere compilata in un altro modo che riflette il significato dato nella specifica del linguaggio.

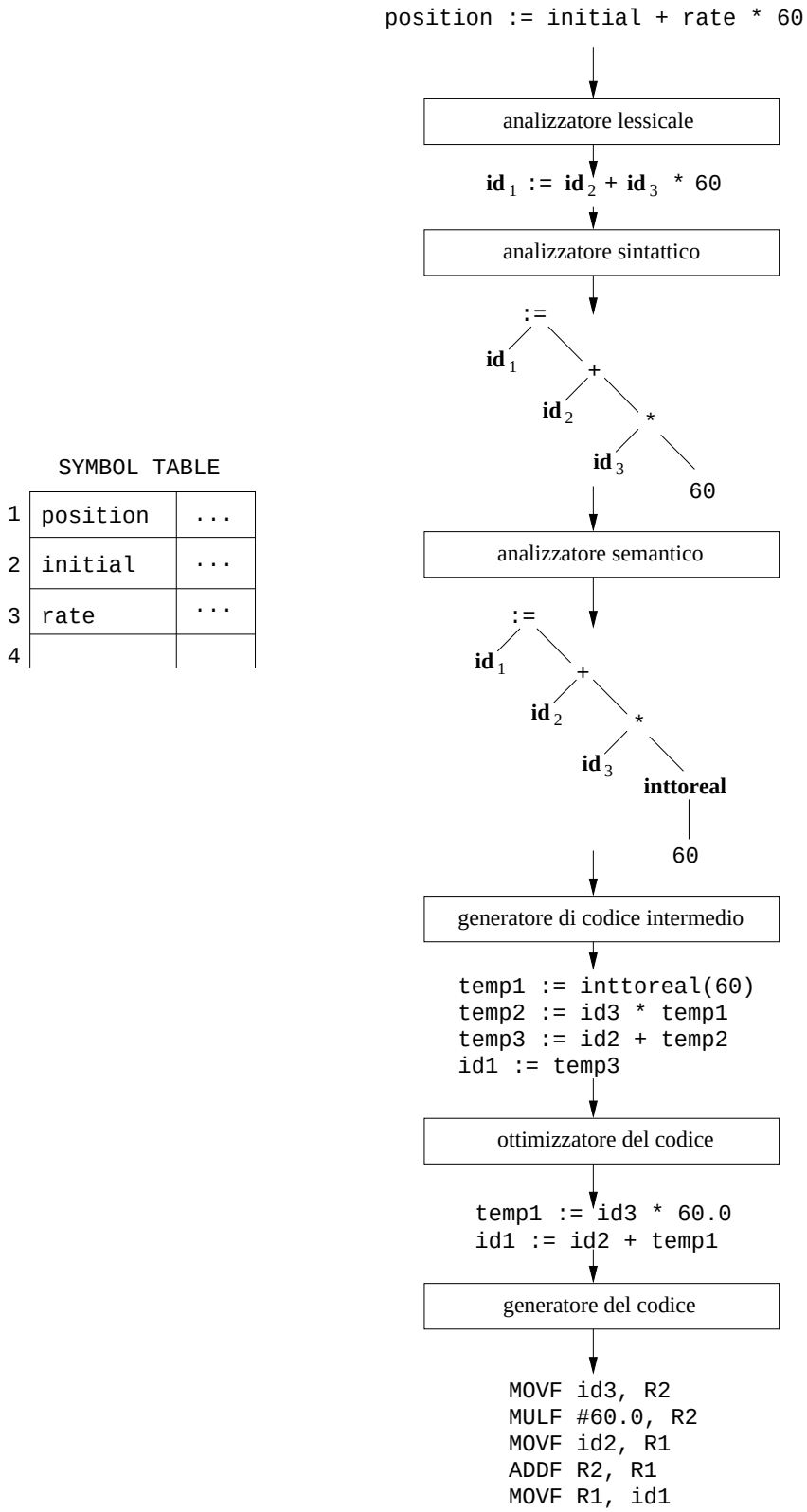

Figura 1.6: Traduzione di uno statement.

ospite verso cui stiamo compilando. Infatti le proprietà principali di questo linguaggio dovrebbero essere le seguenti: deve essere abbastanza semplice da produrre e anche abbastanza semplice da tradurre nel linguaggio della macchina ospite.

Esistono diversi tipi di rappresentazioni intermedie. Noi ne useremo una chiamata "codice a tre indirizzi" (three-address code) che è simile ad un linguaggio assembly di una macchina a registri in cui una cella di memoria può essere usata, nelle operazioni, allo stesso modo di un registro della CPU. Il codice a tre indirizzi consiste di sequenze di istruzioni ognuna delle quali ha al massimo tre operandi. In Figura 1.6 è indicato il codice a tre indirizzi generato per la stringa di esempio:

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

Questo particolare tipo di forma intermedia ha le seguenti caratteristiche:

1. Ogni istruzione ha al più un operatore in aggiunta all'assegnamento. In questo modo, nel generare queste istruzioni, il compilatore deve decidere l'ordine in cui vengono fatte le operazioni (la moltiplicazione precede l'addizione nell'esempio)

2. Il compilatore deve generare un nome temporaneo per denotare il valore calcolato da ogni istruzione

3. Alcune istruzioni hanno meno di tre operandi (ad esempio la prima e l'ultima della traduzione della nostra stringa di esempio)

Nel Capitolo 4 vedremo degli algoritmi per generare il codice intermedio a partire da alcuni costrutti tipici dei linguaggi di programmazione.

1.3.5 Ottimizzazione del codice

Questa fase cerca di migliorare il codice intermedio in modo da renderlo più veloce quando sarà eseguito sulla macchina ospite.

Alcune ottimizzazioni sono banali: ad esempio il codice generato con un semplice algoritmo che traduce automaticamente i costrutti a partire dall'albero sintattico risultante dall'analizzatore semantico (ad esempio il codice prodotto per la nostra stringa di esempio) contiene sicuramente dei nomi temporanei che non sono necessari.

Un esempio lampante è l'ultima istruzione dell'esempio (`id3 := temp3`). Questa è stata generata perchè il nome `temp3` è stato creato come riferimento per il risultato del sotto-albero destro dell'albero sintattico principale.

Inoltre alcune operazioni che sono indicate nell'albero sintattico possono essere fatte una volta per tutte a tempo di compilazione e quindi non devono essere tradotte. È il caso ad esempio dell'operazione `inttoreal(60)`.

Quindi il codice intermedio può essere trasformato nel seguente codice equivalente:

```
temp1 := id3 + 60.0
id1 := id2 + temp1
```

Ci sono moltissimi tipi di ottimizzazione che un compilatore può fare e si ha quasi sempre che il tempo di calcolo maggiore di tutta la compilazione viene speso in questa fase. In ogni caso ci sono diverse semplici ottimizzazioni che non rallentano in maniera eccessiva la compilazione, ma per contro riescono a velocizzare significativamente il tempo di esecuzione del programma target. Ne vedremo dei cenni nel Capitolo 5.

1.3.6 Generazione del codice

L'ultima fase di un compilatore è quella della generazione del codice target (della macchina ospite) che normalmente è un codice macchina (oppure assembly) rilocabile.

Celle di memoria in uno spazio logico sono assegnate ad ogni variabile del programma e il codice intermedio ottimizzato viene tradotto nel linguaggio della macchina ospite. Un aspetto cruciale di questa traduzione è l'assegnazione dei valori delle variabili ai registri. Ad esempio, usando due registri della CPU (R1 ed R2), il codice ottimizzato del nostro esempio diventa il seguente (come era già stato indicato in Figura 1.6):

```
MOVF id3, R2
MULF \#60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

In questo codice assembly i due operandi, separati da virgola, rappresentano rispettivamente una sorgente e una destinazione. La lettera `F` alla fine del nome mnemonico delle istruzioni indica che gli operandi sono numeri reali in rappresentazione floating-point. La prima istruzione copia il valore

contenuto nella variabile `id3` nel registro 2 e poi lo moltiplica con la costante floating-point `60.0`[2]. Il risultato delle operazioni aritmetiche fra due registri viene posto nel registro destinazione. L'istruzione successiva carica nel registro 1 il valore della variabile `initial` e il risultato intermedio precedente viene aggiunto a quest'ultimo valore. Infine il risultato viene posto in `id1`, cioè la zona di memoria dove è stato allocato spazio per la variabile `position`.

Vedremo meglio questa fase nel Capitolo 5.

[2]il fatto che è una costante è indicato dal carattere speciale `#` e il fatto che è floating-point dal punto decimale.