Fasi di un Compilatore

Maria Rita Di Berardini

Dipartimento di Matematica e Informatica
Università di Camerino

Cos'è un compilatore

- Un implementazione compilativa di un linguaggio di programmazione è realizzata tramite un programma che prende il nome di compilatore
- Un compilatore prende in input un dato programma scritto in un linguaggio sorgente e restituisce un programma funzionalmente equivalente scritto in linguaggio target (linguaggio della macchina intermedia scelta per l'implementazione)

Cos'è un compilatore

- La scrittura di un compilatore è un compito abbastanza complesso
- La conoscienza di tecniche efficaci per la scrittura di un compilatore è oggi molto vasta e strutturata
- La scrittura del primo compilatore Fortran (fine anni '50) ha richiesto 18 anni di lavoro da parte di uno staff
- Sono state sviluppate delle tecniche sistematiche per lo sviluppo delle più importanti fasi del compilatore
- Linguaggi di programmazione (es. C), ambienti di programmazione e tools (generazione automatica a partire dalle specifiche delle varie parti del linguaggio sorgente)

Processo di compilazione

- È concettualmente suddiviso in due parti: **analisi** (front-end) e **sintesi** (back-end)
- La **fase di analisi** si occupa di dare una struttura al codice e creare una sua rappresentazione intermedia: le operazioni indicate nel programma sorgente vengono identificate e raggruppate in una struttura ad albero (syntax tree)
- **La fase di sintesi** genera il codice nel linguaggio target a partire dalla rappresentazione intermedia costruita nella fase procedente

La fase di Analisi

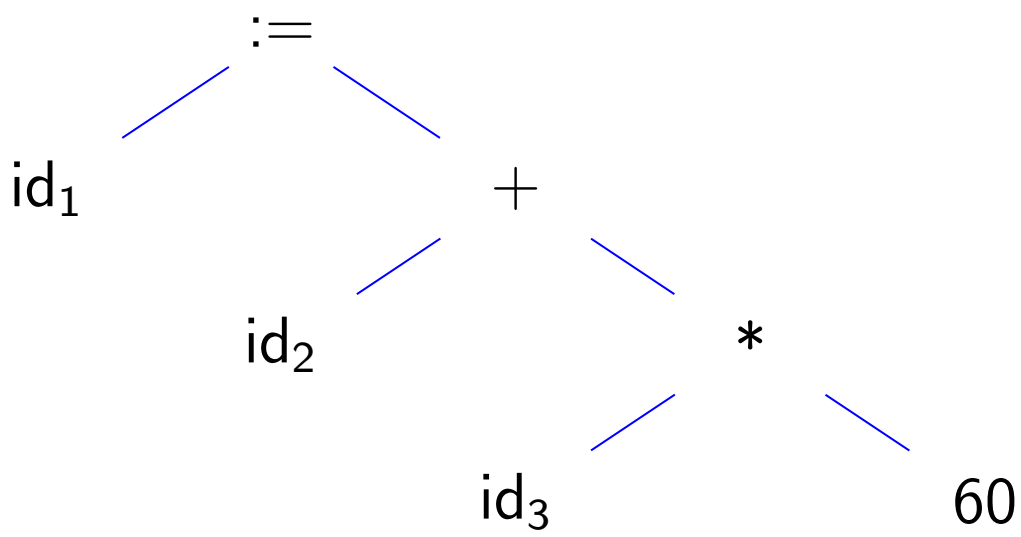
- **Analisi lineare** (**analisi lessicale**, **scanning**): lo stream di caratteri che costituisce il programma in input viene letto e raggruppato in sequenze di caratteri con significato comune detti **token**
  - identificatori, parole chiave (**if**, **while**, **int**), operatori, simboli di punteggiatura
  - cerchiamo le parole del nostro linguaggio
- **Analisi gerarchica** (**analisi sintattica**, **parsing**): i token identificati nella fase precedente vengono raggruppati gerarchicamente tramite delle strutture ad albero che specificano la struttura sintattica delle frasi del programma
  - cerchiamo di mettere insieme le parole identificate in fase di analisi lineare per formare parole sintatticamente corrette
- **Analisi semantica**: in questa fase si verifica che le varie parti del programma siano consistenti l'una con l'altra, a seconda del loro significato

Analisi Lessicale

- Consideriamo, la sequenza di caratteri $\mathbf{id}_1 := \mathbf{id}_2 + \mathbf{id}_3 * 60$
- Lo scanning di questa sequenza identifica i seguenti token:
  1. l'identificatore $\mathbf{id}_1$
  2. l'operatore $:=$
  3. l'identificatore $\mathbf{id}_2$
  4. l'operatore $+$
  5. l'identificatore $\mathbf{id}_3$
  6. l'operatore $*$
  7. il numero 60
- La sequenza di caratteri che corrisponde ad un certo token è detta lessema: in generale ad un dato token (**id**) corrisponono più lessemi ($\mathbf{id}_1$, $\mathbf{id}_2$ e $\mathbf{id}_3$)
- Blank, tabulazioni, newline, e commenti (utili al programmatore ma ignorati dal compilatore) vengono eliminati durante la fase di analisi lessicale

Analisi Sintattica

- I token individuati durante la fase di scanning vengono raggruppati in frasi grammaticali
- Le frasi grammaticali sono rappresentate mediante syntax-tree (alberi sintattici, alberi di derivazione): sulle foglie troviamo gli operandi, sui nodi interni le operazioni



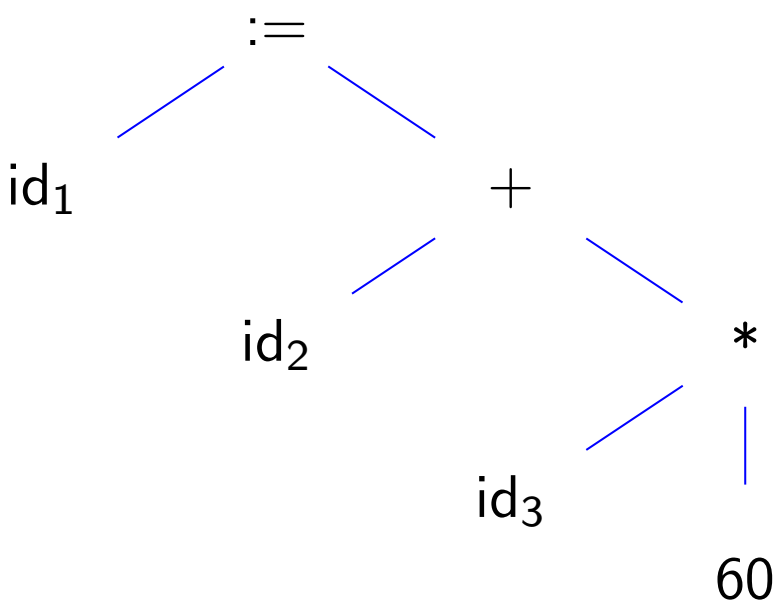
Analisi lessicale vs Analisi sintattica

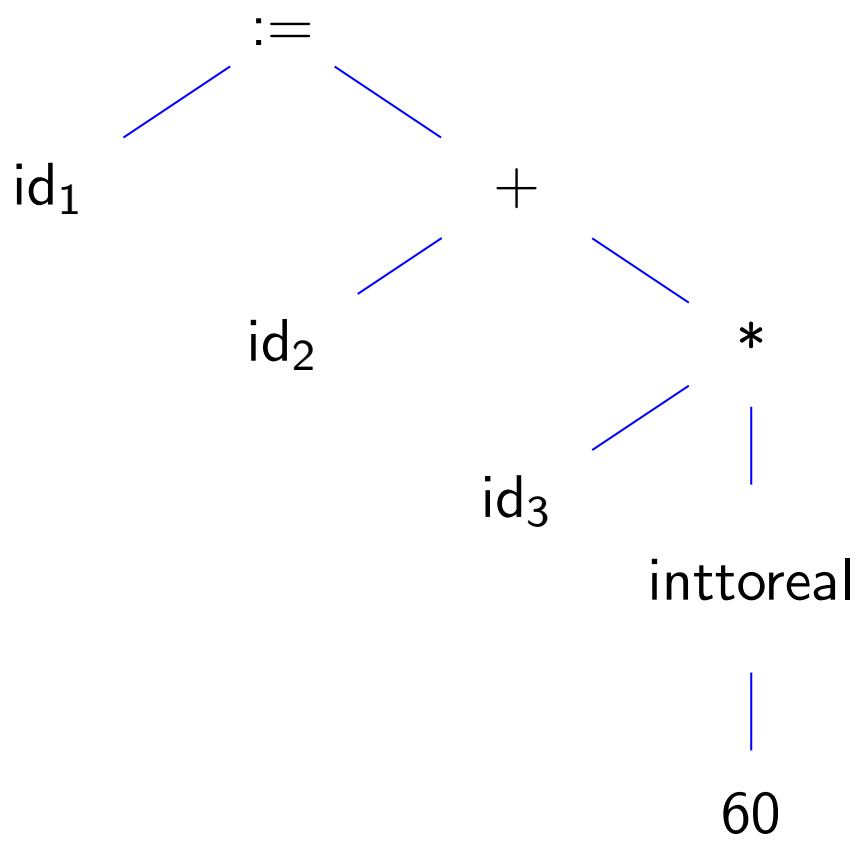
- La divisione tra analisi lessicale e analisi sintattica è in qualche modo arbitraria, dipende essenzialmente dalle scelte del progettista
- L'obiettivo finale è quello di semplificare per quanto possibile la fase di analisi sintattica
- Un fattore discriminante può essere la ricorsione
- Per identificare i token del linguaggio in genere è sufficiente una scansione lineare del programma in input
  - Per riconoscere identificatori (stringhe che cominciano con una lettere e continuano con lettere e/o numeri) basta riconocere la prima lettera e continuare a leggere finchè non si incontra un carattere che non è nè una lettera nè un numero
- La scansione lineare non è abbastanza potente per poter identificare espressioni e statements (costrutti tipicamente ricorsivi)
  - Non si possono associare in maniera corretta i begin e gli end dei costrutti se non si da una struttura gerarchica o annidata all'input

Analisi Semantica

- Controlla che non ci siano errori semantici del programma e acquisisce informazioni sui tipi che verrà usata nella fase di generazione del codice
- La rappresentazione intermedia viene usata per identificare operatori ed operandi di espressioni e statements
- Un componente importante è la fase di type checking
- Durante questa fase, il compilatore verifica che ogni operatore sia applicato ad operandi consentiti dalla specifica del linguaggio
- Può dar luogo ad:
  - errori: uso di numeri reali per indicizzare array
  - conversioni di tipo: se, nell'espressione $\mathbf{id}_1 := \mathbf{id}_2 + \mathbf{id}_3 * 60$, $\mathbf{id}_3$ è un numero reale, l'intero 60 viene automaticamente convertito in un reale
  - in questo caso un nuovo nodo viene aggiunto al syntax tree

Analisi Semantica





Processo di compilazione

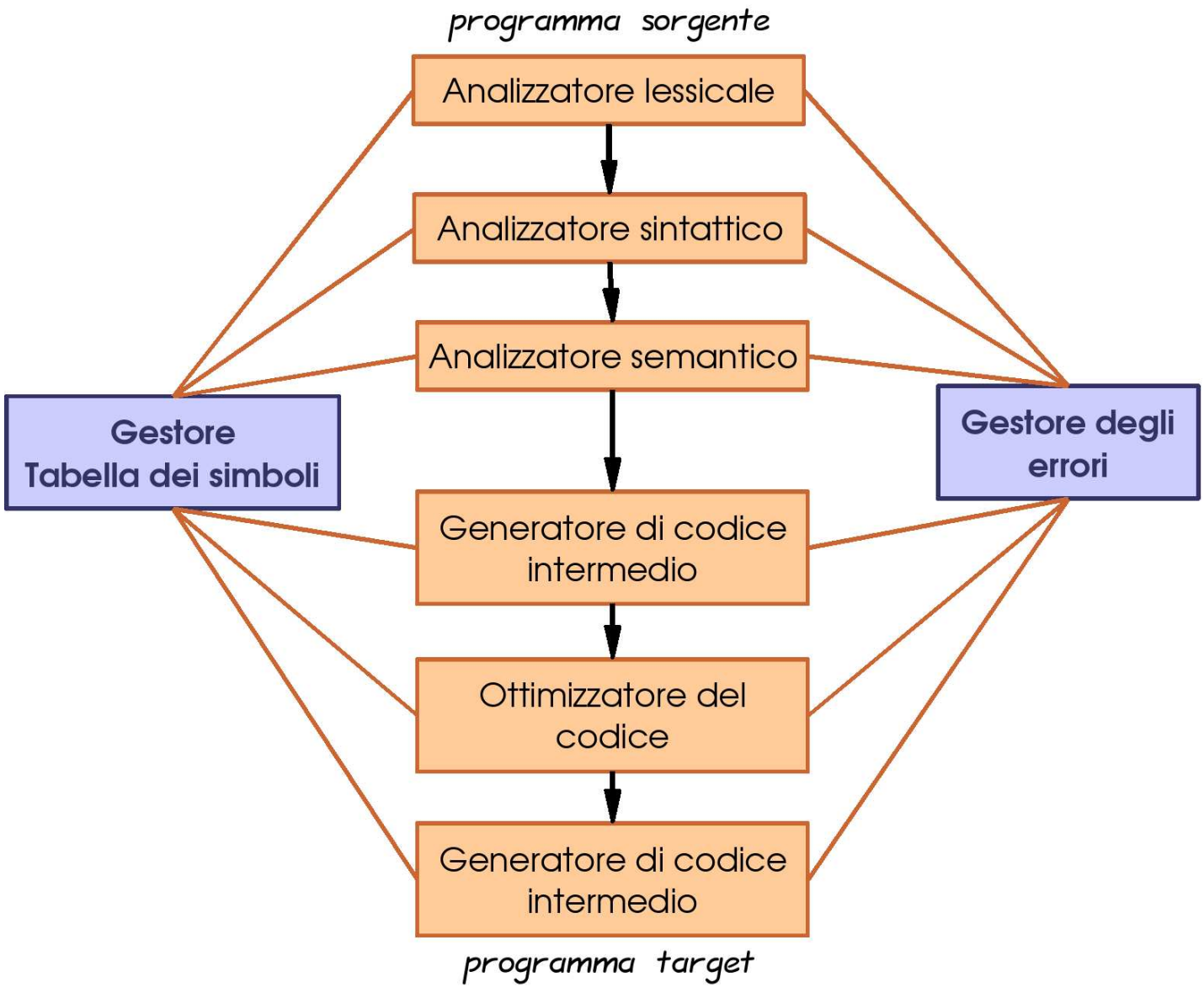


Tabella dei simboli

- Uno dei ruoli fondamentali di un compilatore è quello di memorizzare gli identificatori usati nel programma e relativi attributi
  - valore, tipo, scope, ...
  - per nomi di procedure: il numero e il tipo dei parametri, la modalit di passaggio dei parametri, il tipo di ritorno (se esiste), ...
- La tabella dei simboli è una struttura dati che contiene un **record** per ogni identificatore i cui campi vengono usati per memorizzare i diversi attributi di ciascun identificatore
- Scopo: reperire tutte le informazioni relative ad un identificatore
- Ogni nuovo identificatore identificato durante la fase di analisi lessicale viene aggiunto alla tabella dei simboli
  - Non tutti gli attributi possono essere identificati in fase di scanning
  - Informazioni relative ai tipi: analisi semantica
  - Informazioni relative al numero di byte allocati: generazione del codice

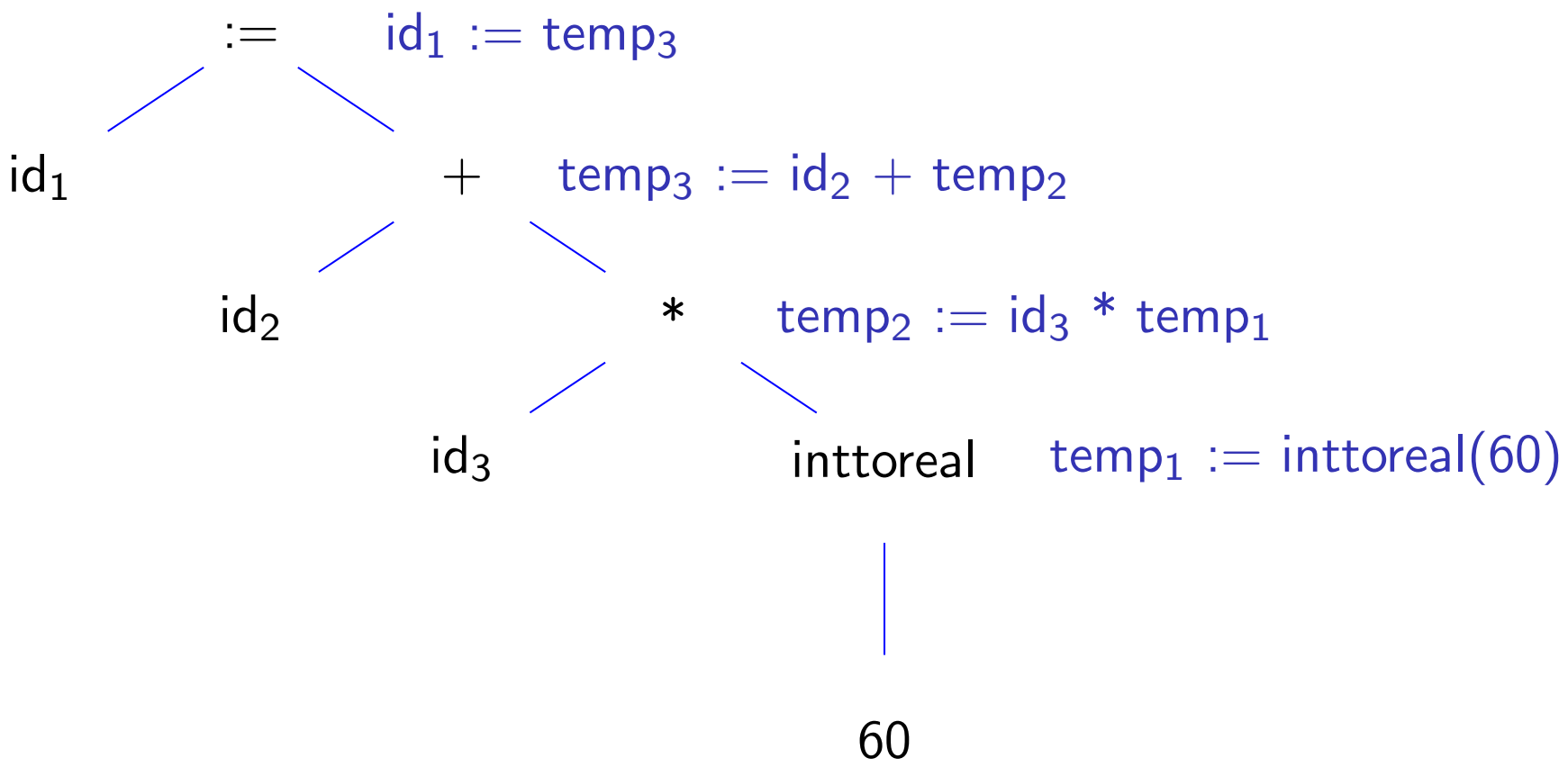
Gestione degli errori

- Altro ruolo fondamentale è il riconoscimento e segnalazione degli errori
- Ogni fase è in grado di rilevare una particolare **classe** di errori
  - Analisi lessicale: i caratteri rimanenti non formano nessun token
  - Analisi sintattica: uno stream di token viola le regole sintattiche del linguaggio
  - Analisi semantica: dei costrutti sintatticamente corretti richiedono operazioni non consentite dagli operandi in gioco, es. somma tra un intero ed un array
- Le fasi di analisi lessicale e analisi sintattica sono quelle che devono fronteggiare il maggior numero di errori
- Gli errori devono essere gestiti in modo da consentire il proseguimento del processo di compilazione: immaginate un compilatore che si blocca ogni volta che omettete un ;

Generazione del codice intermedio

- Alcuni compilatori, al termine della fase di analisi, generano una rappresentazione intermedia esplicita del programma sorgente
- Possiamo pensare a questa rappresentazione come ad un programma scritto nel linguaggio di una macchina astratta poco distante dalla macchina ospite verso cui stiamo compilando
- Le principali proprietà di questo linguaggio sono le seguenti:
  - semplice da produrre
  - semplice da tradurre nel linguaggio della macchina ospite
- Esistono diverse rappresentazioni intermedie, noi useremo il *codice a tre indirizzi* (three-address-code)

Generazione del codice intermedio



Generazione del codice intermedio

- Generatore prende in input l'albero di derivazione generato durante la fase di analisi semantica e restituisce il seguente three-address code

$$\begin{aligned} \texttt{temp}_1 &:= \texttt{inttoreal(60)} \\ \texttt{temp}_2 &:= \texttt{id}_3 * \texttt{temp}_1 \\ \texttt{temp}_3 &:= \texttt{id}_2 + \texttt{temp}_2 \\ \texttt{id}_1 &:= \texttt{temp}_3 \end{aligned}$$

- Caratteristiche del three-address code:
  1. ogni istruzione ha al più un operatore oltre all'operatore di assegnamento
  2. viene generato un nome temporaneo per denotare il valore calcolato da ogni istruzione
  3. alcune istruzioni possono avere un solo operando

Ottimizzazione del codice

- Ottimizzare il codice prodotto in maniera da renderlo più efficiente quando sarà eseguito sulla macchina ospite

$$
\begin{aligned}
&(1) \quad temp_1 := inttoreal(60)\\
&(2) \quad temp_2 := id_3 * temp_1\\
&(3) \quad temp_3 := id_2 + temp_2\\
&(4) \quad id_1 := temp_3
\end{aligned}
$$

(1) e (2) $\Longrightarrow$ $temp_1 := id_3 * 60.0$

(3) e (4) $\Longrightarrow$ $id_1 := id_2 + temp_1$

- Queste semplici ottimizzazioni del codice prodotto
  - Non rallentano il processo di compilazione
  - Riescono a migliorare significativamente il tempo di esecuzione del programma

Ottimizzazione del codice

- Genera il codice target a partire dal codice intermedio ottimizzato
- Asumendo che il linguaggio target sia il codice assembly

$$\texttt{temp}_1 \texttt{ := id}_3 \texttt{ * 60.0}$$
$$\texttt{id}_1 \texttt{ := id}_2 \texttt{ + temp}_1$$

```
MOV id3, R2
MUL i#60.0, R2
MOV id2, R1
ADD R2, R1
MOV R1, id1
```