Dispensa 6

6.1 Generazione del codice intermedio e Codice a 3-indirizzi: generalità

Dopo le fasi di analisi sintattica e analisi semantica il compilatore genera un'esplicita *rappresentazione intermedia* del programma sorgente. Esistono diversi linguaggi usati nella fase di generazione del codice intermedio, tra questi studieremo un linguaggio chiamato codice a 3-indirizzi.
Per iniziare a illustrare il formato di una istruzione di codice a 3-indirizzi facciamo riferimento al generico statement di assegnamento che è della forma generale:

```
x := y OP z
```

qui:
OP è un generico operatore binario (per es. logico o aritmetico)
x, y, z sono nomi di costanti o variabili predefinite o generati dal compilatore (in questo caso si chiamano *temporanei*).

La necessità di temporanei, cioè di variabili generate dal compilatore e non presenti nel programma sorgente, è dovuta al fatto che nella parte destra di uno statement ci può essere solo un operatore, e quindi espressioni complesse vanno spezzate in sequenze di statements. Per esempio, la stringa sorgente:

```
x + y * z
```

è spezzata nella sequenza:

```
t1 := y * z
t2 := x + t1
```

dove il compilatore ha dovuto generare i temporanei t_1 e t_2 (sostanzialmente delle variabili d'appoggio) per mantenere il valore computato da ogni istruzione.

Nota: Il nome codice a *3-indirizzi* del linguaggio deriva dal fatto che uno statement contiene sempre al più tre indirizzi di memoria per le variabili coinvolte.

6.2 Codice a 3-indirizzi: sintassi

Le più comuni istruzioni del codice a 3-indirizzi sono:

- *statement di assegnamento*

è della forma:

```
x := y OP z
```

oppure della forma:

```
x := OP z
```

dove nel primo caso OP è un operatore binario aritmetico o logico, mentre nel secondo caso OP è un operatore unario (per es. il meno aritmetico o il not logico). L'istruzione di assegnamento assegna ad x il valore del risultato dell'operazione applicata a y e z (nel primo caso) oppure a z (nel secondo caso).

- *statement di copia*

è della forma:

x := y

L'istruzione di copia assegna ad x il valore di y.

- *statement di salto incondizionato*

è della forma:

goto L

L'istruzione indica che il prossimo statement da eseguire è quello labellato L. Infatti le istruzioni in codice a 3-indirizzi hanno associate delle label simboliche, un po' come avviene nei linguaggi assembler-like.

- *statement di salto condizionato*

è della forma:

if x RELOP y goto L

dove RELOP è un operatore condizionale del tipo <, >, =, ≤, etc. L'istruzione indica che il prossimo statement da eseguire è quello labellato L nel caso in cui la relazione x RELOP y è vera, altrimenti si procede col successivo statement nel codice.

Nota importante: La tecnica più comune e più usata nei compilatori per implementare il codice a 3-indirizzi è quella di memorizzare le istruzioni in *quadruple*, ossia record con quattro campi OP, ARG1, ARG2, RESULT. In generale, il campo OP è usato per l'operatore, i campi ARG1 e ARG2 per i suoi argomenti, il campo RESULT per il risultato; statements con operatori unari non utilizzano il campo ARG2, mentre salti condizionati e incondizionati utilizzano il campo RESULT per la label. Alternativamente, il codice a 3-indirizzi può essere implementato attraverso *triple*, ossia record con tre campi in cui è assente il campo RESULT. In questo caso si fa riferimento al risultato di una operazione direttamente attraverso il puntatore alla tripla che calcola quel risultato.

Esempio1:
Si consideri la seguente stringa sorgente per un'istruzione di assegnamento:

a := b * -c + b * -c

la sua rappresentazione in codice a 3-indirizzi è:

t_1 := -c
t_2 := b * t_1
t_3 := -c
t_4 := b * t_3
t_5 := t_2 + t_4
a := t_5

Nota: in questo esempio abbiamo omesso le label simboliche associate agli statements in quanto non utilizzate.

Esempio2:
Il seguente frammento di programma in codice a 3-indirizzi permette il calcolo del valore assoluto di x:

```
100: if x >= 0 goto 103
101: t1 := -x
102: goto 104
103: t1 := x
104: …
```

Come si può facilmente verificare, il programma proseguirà dall'istruzione labellata 104 sempre con il corretto valore assoluto di x mantenuto nel temporaneo t_1.

6.3 Caso di studio 1: sintassi tipata

Nel caso in cui il linguaggio sorgente è tipato, la sintassi del codice a 3-indirizzi deve essere leggermente modificata al fine di gestire i tipi di dato. Restringiamo la nostra attenzione a un semplice caso di studio relativo ad un contesto aritmetico che include i soli tipi di dato integer e real e che permette la coercizione da interi a reali.
Ad esempio, se consideriamo in questo contesto l'istruzione per la somma:
`x + y`
(la quale senza tipi sarebbe stata tradotta in codice a 3-indirizzi dallo statement t_1 := x + y) essa è rappresentata in codice a 3-indirizzi dalla sequenza di statements (assumendo x intero e y reale):

```
t1 := int_to_real x
t2 := t1 real+ y
```

In sostanza, si usa prima un nuovo operatore unario int_to_real per trasformare il valore intero di x e metterlo nel temporaneo t_1. Dopodiché si usa l'operatore esteso real+ per la somma che specifica l'addizione fra reali.
Pertanto la sintassi tipata estende quella tradizionale del codice a 3-indirizzi semplicemente aggiungendo il nuovo operatore int_to_real per effettuare la coercizione da interi a reali, e usando i tradizionali operatori aritmetici che vengono estesi però col prefisso real oppure int (quindi int+ , real+ , int *, real *, etc.) per specificare se l'operazione è applicata a interi o reali.

Esercizio: (tratto da traccia d'esame)
Fornire la rappresentazione intermedia in codice a 3-indirizzi per la stringa sorgente `x := y * z` assumendo che y sia di tipo 'integer' e x e z di tipo 'real'.

Soluzione:
La sequenza di statements corrispondente alla stringa sorgente data è:

```
t1 := int_to_real y
t2 := t1 real* z
x := t2
```

Esercizio: (tratto da traccia d'esame)
Fornire la rappresentazione intermedia in codice a 3-indirizzi per la stringa sorgente `a + b - c` assumendo che `a` e `b` siano reali e `c` intero, e `+` con precedenza maggiore rispetto a `-`

Soluzione:
La sequenza di statements corrispondente alla stringa sorgente data è:
t_1 := a real+ b
t_2 := int_to_real c
t_3 := t_1 real- t_2

6.4 Caso di studio 2: espressioni booleane e relazionali

In analogia al caso delle espressioni aritmetiche, le espressioni booleane sono gestite facilmente tenendo presente l'associatività e la precedenza degli operatori logici. Ad esempio, la stringa sorgente:

a OR b AND NOT c

sarà rappresentata dalla sequenza di statements:

t_1 := not c
t_2 := b AND t_1
t_3 := a OR t_2

Le espressioni relazionali (ossia contenenti gli operatori <, >, =, ≤, etc.) vanno gestite tenendone presente la sottintesa semantica. Per fare un semplice esempio, l'espressione relazionale:

a < b

ha come significato sottinteso (assumendo che 1 significa 'vero' e 0 significa 'falso'):

if a < b then 1 else 0

e pertanto può essere tradotta nella sequenza di statements in codice a 3-indirizzi:

100: if a < b goto 103
101: t_1 := 0
102: goto 104
103: t_1 := 1
104: ...

In questo modo nel continuo del codice dall'istruzione labellata 104 l'espressione relazionale sarà sempre gestita col suo corretto valore vero o falso propriamente mantenuto nel temporaneo t_1.

Esercizio: (tratto da traccia d'esame)
Fornire la rappresentazione intermedia in codice a 3-indirizzi per la stringa sorgente:

a OR (b > c)

Soluzione:
La sequenza di statements corrispondente alla stringa sorgente data è:

100: if b > c goto 103
101: t_1 := 0
102: goto 104
103: t_1 := 1
104: t_2 := a OR t_1

Esercizio: (tratto da traccia d'esame)
Fornire la rappresentazione intermedia in codice a 3-indirizzi per la stringa sorgente:

a < (b + 8) AND c

Soluzione:
La sequenza di statements corrispondente alla stringa sorgente data è:

100: t_1 := b + 8
101: if a < t_1 goto 104
102: t_2 := 0
103: goto 105
104: t_2 := 1
105: t_3 := t_2 AND c

Esercizio: (tratto da traccia d'esame)
Fornire la rappresentazione intermedia in codice a 3-indirizzi per la stringa sorgente:

x < y AND NOT z

Soluzione:
La sequenza di statements corrispondente alla stringa sorgente data è:

100: if x < y goto 103
101: t_1 := 0
102: goto 104
103: t_1 := 1
104: t_2 := NOT z
105: t_3 := t_1 AND t_2

6.5 Generazione di codice intermedio tramite schemi di traslazione

Per concludere, di seguito mostriamo alcuni casi di studio per la generazione di codice a 3-indirizzi tramite traduzione guidata dalla sintassi. Come visto nella teoria degli schemi di traslazione, per

generare correttamente il codice output di una qualsiasi stringa sorgente bisognerà eseguire le azioni semantiche nell'ordine in cui esse appaiono in una visita depth-first del parse tree di quella stringa.

Caso di studio 1: Schema di traslazione per la generazione di codice a 3-indirizzi per istruzioni di assegnamento ed espressioni artitmetiche

$S \rightarrow$ id := E	{ EMIT(id.val ':=' E.val); }
$E \rightarrow E_1 + E_2$	{ E.val := NEWTEMP(); EMIT(E.val ':=' E_1.val '+' E_2.val); }
$E \rightarrow E_1 * E_2$	{ E.val := NEWTEMP(); EMIT(E.val ':=' E_1.val '*' E_2.val); }
$E \rightarrow -E_1$	{ E.val := NEWTEMP(); EMIT(E.val ':=' '-'E_1.val); }
$E \rightarrow (E_1)$	{ E.val := E_1.val ; }
$E \rightarrow$ id	{ E.val := id.val ; }

Si noti che nello schema:

- val è l'attributo usato per mantenere il valore del non-terminale E e del token id

- NEWTEMP() è la funzione per generare temporanei, e ritorna una sequenza di nomi distinti t_1, t_2, … in risposta a sue successive chiamate

- EMIT è la funzione che emette statements in codice a 3-indirizzi su file output. Si assume che EMIT stampa inalterate costanti ed espressioni racchiuse tra apici, altrimenti valuta un'espressione e ne stampa il valore. Ad esempio, EMIT(x ':=' y '+' z) stampa lo statement x := y + 3 nel caso in cui x e y sono costanti e z è un'espressione con valore 3.

Caso di studio 2: Schema di traslazione per la generazione di codice a 3-indirizzi e type checking con coercizione da interi a reali per l'istruzione di somma aritmetica

$E \rightarrow E_1 + E_2$ { E.val := NEWTEMP();
if E_1.type = integer and E_2.type = integer then begin
EMIT(E.val ':=' E_1.val 'int+' E_2.val);
E.type := integer;
end
else if E_1.type = real and E_2.type = real then begin
EMIT(E.val ':=' E_1.val 'real+' E_2.val);
E.type := real;
end
else if E_1.type = integer and E_2.type = real then begin
u := NEWTEMP();
EMIT(u ':=' 'int_to_real' E_1.val);
EMIT(E.val ':=' u 'real+' E_2.val);

```
        E.type := real;
    end
else if E1.type = real and E2.type = integer then begin
        u := NEWTEMP();
        EMIT(u ':=' 'int_to_real' E2.val);
        EMIT(E.val ':=' E1.val 'real+' u);
        E.type := real;
    end
    else
        E.type := type_error;  }
```

Caso di studio 3: Schema di traslazione per la generazione di codice a 3-indirizzi per espressioni booleane e relazionali

$E \rightarrow E_1$ OR E_2	{ E.val := NEWTEMP(); EMIT(E.val ':=' E_1.val 'OR' E_2.val); }
$E \rightarrow E_1$ AND E_2	{ E.val := NEWTEMP(); EMIT(E.val ':=' E_1.val 'AND' E_2.val); }
$E \rightarrow$ NOT E_1	{ E.val := NEWTEMP(); EMIT(E.val ':=' 'NOT'E_1.val); }
$E \rightarrow (E_1)$	{ E.val := E_1.val ; }
$E \rightarrow id_1$ RELOP id_2	{ E.val := NEWTEMP(); EMIT('if' id_1.val RELOP.val id_2.val 'goto' *nextstat* + 3); EMIT (E.val ':=' '0'); EMIT ('goto' *nextstat* + 2); EMIT (E.val ':=' '1'); }
E → true	{ E.val := NEWTEMP(); EMIT(E.val ':=' '1'); }
E → false	{ E.val := NEWTEMP(); EMIT(E.val ':=' '0'); }

Si noti che in questo caso RELOP indica un operatore relazionale, la variabile *nextstat* contiene l'indice del prossimo statement nella sequenza output, e la funzione EMIT ha come effetto collaterale quello di incrementare *nextstat* dopo aver prodotto uno statement.