RELAZIONE DI LFC

GRUPPO 61:

GRIP
JOCO
MAROK
MIOPIO
NICHEL
SPRANGA

COMPILATORE

COMPILAZIONE DEI SORGENTI C

Per la compilazione dei sorgenti C viene utilizzato il compilatore unix GCC sulle macchine MORGANA e GAMMA
Inoltre vengono usati FLEX e BISON al posto di LEX e YACC per elaborare le specifiche lessicali e sintattiche.

USO DEL COMPILATORE

Il compilatore viene lanciato con il comando:
PARSER [-d/-s] < nome_file_sorgente > nome_file_destinazione.
L'utente può scegliere tra le opzioni -s, che visualizza la symbol table, e -d che visualizza i passi di parsificazione del file sorgente, il cui nome dev'essere preceduto dal simbolo "<" per la ridirezione dell'input, mentre il nome del file destinazione dev'essere preceduto dal simbolo ">" per la ridirezione dell'output.

COMPILATORE A STRATI

LIVELLO 1

Il primo strato del compilatore è stato progettato per riconoscere espressioni numeriche con somme, prodotti, sottrazioni e divisioni.

COMPONENTI DEL COMPILATORE

A questo livello il compilatore è composto da 2 moduli C, una specifica lessicale EXPR.L e una specifica sintattica SOURCE.Y.
I moduli C si chiamano rispettivamente:

GENERA.C è il modulo di generazione del codice intermedio sotto forma di quadruple.

SYM_TBL.C dove vengono descritte le funzioni di gestione della Symbol Table.

SPECIFICA SINTATTICA:

```
%union {
      numero dval;
      coppia *espr;
      int op;    }

%token <op> ADDOP
%token <op> MULOP
%token <dval> INTERO
%token <dval> REALE

%type <espr> smpl_expr
%type <espr> term
%type <espr> factor
%type <op> sign
%%
Start : smpl_expr ';' { printf("HALT\n$end\n");
                  exit(0);}
       ;
smpl_expr   : term  {$$ = $1;}
         | sign term { if (is_coppia_int($1))
                     $$ = newtemp(INTERO);
                  else $$ = newtemp(REALE);
                  if ($1 == MENO) emit(SEGNO,$2,NULL,NULL); }
         | smpl_expr ADDOP term  {
                if ((is_coppia_int($1))
                 && (is_coppia_int($3)))
                 $$ = newtemp(INTERO);
               else $$ = newtemp(REALE);
                        emit($2,$$,$1,$3); }
         ;
sign      : ADDOP { $$ = $1; }
         ;
term    : term MULOP factor  {if ((is_coppia_int($1)) && (is_coppia_int($3)))
                        $$ = newtemp(INTERO);
                     else $$ = newtemp(REALE);
                     emit($2,$$,$1,$3); }
      | factor           {$$ = $1; }
      ;
factor  : REALE              {$$ = const2cop(REALE, $1.reale); }
      | INTERO             {$$ = const2cop(INTERO, $1.intero); }
      | '(' smpl_expr ')'    {$$ = $2;}
      ;
```

Abbiamo definito i seguenti tipi come attributi, DVAL serve per contenere le costanti intere e reali, ESPR è un puntatore a coppie, che serve per contenere costanti e temporanei, OP è un intero e serve per contenere il tipo dell'operatore, ad esempio distingue, nel caso venga identificato un token ADDOP, il "-" dal "+", assumendo valore *meno* o *più.*

Quando viene riconosciuta una somma tra due termini viene generato un nuovo temporaneo per contenere il valore della somma, tenendo conto se l'istruzione opera tra interi o reali, e viene generata la quadrupla.

GENERA.C

In questo modulo sono incluse le seguenti definizioni di tipi e funzioni:

```
typedef union {
    int intero;
    float reale; } numero;
```

Il tipo definito numero serve per contenere i numeri interi o reali passati dalla specifica lessicale che servono per contenere le costanti.

```
typedef struct {
   int tipo;
   numero valore; } coppia;
```

La coppia può contenere le costanti intere e reali ed è costituita da due campi in modo da poter distinguere quando viene istanziata una costante intera o una reale.

```
numero get_val_cop(coppia *temp);
```

Serve per prendere il campo "valore" della coppia

```
int get_tip_cop(coppia *temp);
```

Serve per prendere il campo "tipo" della coppia.

```
void emit(int tipo,int op,coppia *t,coppia
*x,coppia *y);
```

La emit stampa su schermo una quadrupla. Richiede come parametri il tipo, cioè *addop* o *mulop*, e l'operatore vero e proprio (può essere il + o il - per l'*addop*, * o / per la *mulop*)

```
coppia *newtemp();
```

Alloca lo spazio per un nuovo temporaneo e lo restituisce come valore. È indispensabile per mantenere i valori intermedi delle operazioni delle espressioni.

```
coppia *const2cop(int tipo, numero n);
```

Trasforma la costante n in una coppia. A livello della specifica sintattica vogliamo trattare tutti i tipi di dati, anche futuri, (quindi variabili e procedure) come coppie.

SYM_TBL.C

In questo modulo sono incluse le seguenti definizioni di tipi e funzioni:

```
typedef struct {
      char  *nomevar;
      int   tipo;
      int   livello;
      int   offset; } s_tab;
```

Ogni elemento della symbol table è costituito da un oggetto di tipo "s_tab" con quattro campi relativi al nome, tipo, livello ed offset della variabile che inseriremo in symbol table.
Quando inseriamo un temporaneo il nome non è specificato e varrà NULL.

```
char *get_nomevar_rec(s_tab *rec);
 int get_tipo_rec(s_tab *rec);
 int get_livello_rec(s_tab *rec);
 int get_offset_rec(s_tab *rec);
```

Utilizziamo funzioni per leggere ogni campo del record; non vi accediamo mai direttamente.

```
void put_nomevar_rec(char *nome, s_tab
*rec);
void put_tipo_rec(int x, s_tab *rec);
void put_livello_rec(int x, s_tab *rec);
void put_offset_rec(int x, s_tab *rec);
```

La stessa cosa vale in scrittura.

```
int push(s_tab *rec);
int pop();
```

La symbol table è gestita come uno stack, a cui possiamo aggiungere un elemento con una *push* e toglierle con una *pop*.

```
s_tab *make_rec (char *str, int tipo, int liv,
int off);
```

Prima di aggiungere un elemento in symbol table dobbiamo crearlo, cioè allocarne lo spazio e scriverne i campi in maniera opportuna.

```
int lookup(char *nome);
```

La lookup ha il compito di recuperare dalla symbol table i dati relativi ad una variabile precedentemente inserita. Specificando il nome di una variabile, infatti, ne restituisce la posizione in symbol table.

LIVELLO 2

Il secondo strato del compilatore è stato progettato per poter dichiarare le variabili e usarle nelle espressioni.

COMPONENTI DEL COMPILATORE

A questo livello il compilatore è composto da 2 moduli C una specifica lessicale EXPR.L e una specifica sintattica SOURCE.Y, esattamente come il livello 1. Per quanto riguarda i moduli C sono stati solo riorganizzati gli header files in modo da avere una sola definizione delle funzioni esterne delle costanti e dei tipi globali. Abbiamo invece dovuto aggiungere delle regole nella speicifica sintattica tali da riconoscere le dichiarazioni di variabili.

SPECIFICA SINTATTICA:

```
%union {
      numero dval;
      coppia *espr;
      int op;
      char *name_id;
     }
```

Abbiamo incluso in questa union l'attributo *char id* che serve per mantenere il nome della variabile.

```
 /* parole chiave riservate */
%token PROGRAM INIZIO END NOT VAR INTEGER REAL RELOP

 /* operatori */
%token <op> ASSIGNOP
%token <op> ADDOP
%token <op> MULOP

 /* variabili e numeri */
%token <name_id> ID
%token <dval> INTERO
%token <dval> REALE

 /* espressioni */
%type <espr> smpl_expr
%type <espr> term
%type <espr> factor
%type <espr> variable
%type <espr> expression

%type <op> identifier_list
```

```
%type <op> declarations
%type <op> type
%type <op> standard_type
%type <op> compound_statement
%type <op> optional_statements
%type <op> statement_list
%type <op> statement

%%
Start : PROGRAM ID ':' declarations
compound_statement { printf (" Fine
compilazione.\n\n");}
    ;
```

```
identifier_list : ID { push ( make_rec( $1,
livello(), INTERO, offset())) ; $$ = 1; }
            | identifier_list ',' ID { push (
make_rec ( $3, livello(), INTERO,
offset())); $$ = $1 + 1; }
            ;
```

Quando viene incontrato un identificatore viene inserito il suo nome in symbol table tramite la push, specificandogli il livello di appartenenza (in questo caso è sempre zero), il tipo e l'offset.
Inoltre vengono contati gli identificatori.

```
declarations : declarations VAR
identifier_list ':' type ';' { metti_tipo_var
( $5, $3); }
          |
          ;
```

Questa istruzione serve per poter inserire il tipo corretto agli ultimi $3 elementi inseriti in symbol table non appena viene trovato dopo il simbolo ":" il type ($5), compito svolto dalla funzione *metti_tipo_var*.

```
type : standard_type   {$$ = $1;}
    ;
```

```
standard_type : INTEGER { $$ =
INTEGER;}
          | REAL { $$ = REAL;}
          ;
```

Vengono riconosciuti i tipi intero e reale.

```
compound_statement : INIZIO
optional_statements END  {}
              ;
```

```
optional_statements : statement_list
              ;
```

```
statement_list : statement ';'
           | statement_list ';' statement ';'
           ;
```

```
statement : variable ASSIGNOP
expression { emit ($2, $1, $3, NULL);
}
       | compound_statement
       ;
```

Viene generata l'istruzione di assegnamento ASSIGN con i suoi operandi.
$1 indica la variabile in cui mettere il risultato, che si trova in $3.

```
variable : ID   { $$ = lookup( $1 );  }
       ;
```

Quando viene riconosciuto un identificatore, assegniamo all'attributo di variable la coppia restituita dalla funzione lookup, che prende come parametro il nome della variabile.
Tale coppia comprende il tipo VARTEMP, che serve per distinguere le variabili dalle costanti, ed un intero, che corrisponde alla posizione di tale variabile in symbol table.

```
expression : smpl_expr
        ;
:
:
:
```

```
factor  : ID              { $$ = lookup
( $1 ) ; }
      | REALE             { $$ = const2cop(REALE, $1.reale); }
      | INTERO            { $$ = const2cop(INTERO, $1.intero); }
      | '(' smpl_expr ')'   { $$ = $2;}
      ;
```

DEF.H

```
typedef union {
    int intero;
    float reale;
    int posTBL; } numero;
:
```

Nella union numero è stato inserito un ulteriore campo posTBL per contenere appunto la posizione in symbol table di una variabile nel secondo campo della coppia, che è appunto di tipo *numero*.

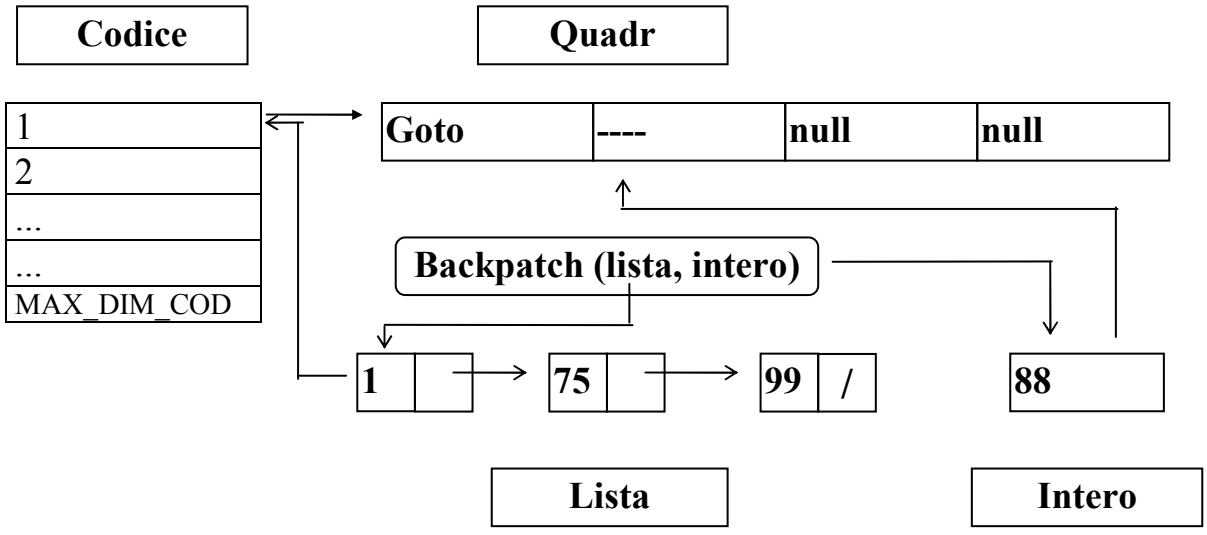
Non sono state definite nuove funzioni e continua a valere il discorso fatto per le funzioni di livello 1.

LIVELLO 3

Il terzo strato è il compilatore quasi completo, mancano soltanto le procedure e comprende, quindi, gli statement (le strutture di *if ... then ... else*, *while ... do*). È quindi stato necessario definire la parte di codice relativa al backpatching.

COMPONENTI DEL COMPILATORE

A questo livello il compilatore è composto da 6 moduli C, una specifica lessicale EXPR.L e una specifica sintattica SOURCE.Y.
Sono stati nuovamente riorganizzati i moduli C in modo da separare meglio le funzioni a seconda del loro compito.
Il nuovo modulo COPPIA.C comprende ora tutte le funzioni di gestione della coppia che erano prima incluse nel modulo GENERA.C, anch'esso modificato per poter permettere il backpatching: la emit, infatti, ora non produce immediatamente la stampa video del codice, che viene immagazzinato in una struttura temporanea, un vettore di quadruple CODICE [].
È stato anche creato un nuovo modulo LISTE.C, che gestisce la creazione e modifica di liste di interi, che verranno utilizzate per contenere la truelist e la falselist del backpatching.
La funzione di backpatching vera e propria è definita nel modulo BACK.C. Prende come parametri una lista ed un intero. La lista di interi rappresenta una lista di istruzioni, dove ad ogni intero è collegata la relativa linea di codice, l'intero è l'operando da inserire in ciascuna istruzione della lista.



SPECIFICA SINTATTICA:

```
%union {
      numero dval;
      struct { coppia *temp;
            lista *truelist;
            lista *falselist;} espr;
      int op;
      int quadrup ;
      char *name_id;
      struct { lista *truelist;
            lista *falselist;} liste;
     }
```

La maggiore differenza è la ridefinizione del tipo ESPR: ora infatti è una struttura composta di tre campi, il primo è un puntatore a coppia come nei precedenti due livelli, sono stati aggiunti il secondo ed il terzo campo per poter avere attributi da associare alle liste truelist e falselist del backpatching.
Vi è inoltre un nuovo tipo LISTE, quando non ho un temporaneo da sintetizzare, e quindi mi bastano i due campi delle liste.

```
 /* parole chiave riservate */
:
:
%token IF THEN ELSE WHILE DO
:
:
%type <op> optional_statements
%type <op> statement_list
%type <liste> statement
%type <quadrup> mark
%type <liste> mark_g

%%
Start : PROGRAM ID ';'
declarations compound_statement { printf
        ("Fine compilazione.\n\n");
        emit(HALT,NULL,NULL,NULL);
                print_symtbl();
                print_cod();}
     ;

optional_statements : statement_list {}
                ;

statement_list : statement ';'  {}
            | statement_list statement ';' {}
            ;
```

Vengono riconosciute le parole chiave della struttura IF THEN ELSE e WHILE DO.

Terminata la compilazione vengono stampati la symbol table e il codice.

Non associo nessuna azione a optional statement e statement list.

```
statement : variable ASSIGNOP expression
{ emit ($2, $1.temp, $3.temp, NULL); }
        | compound_statement  { }
        | IF expression THEN mark statement
mark_g ELSE mark statement mark {
        backpatch($2.truelist,$4);
        backpatch($2.falselist,$8);
        backpatch($6.truelist,$10); }
```

Bisogna fare tre volte backpatch.
Le prime due rispettivamente per la truelist e la falselist dell'expression, la terza per l'else, o meglio per la goto generata dalla mark_g prima dell'else.

```
        | WHILE mark expression DO mark
statement mark_g { backpatch($3.truelist, $5);
        backpatch($3.falselist, get_nextquad());
        backpatch($7.list, $2); }
        ;
```

Anche in questo caso c'è bisogno di tre backpatch: i primi due per l'expression, il terzo per la goto che chiude il while e ritorna alla prima istruzione di expression.

```
expression : smpl_expr  { $$.temp = $1.temp; }
        | smpl_expr RELOP smpl_expr
        {$$.temp=newtemp(tipo_temporaneo($1.
         temp,$3.temp));
        emit($2, $1.temp, $3.temp, NULL);
        $$.truelist = makelist(get_nextquad()-1);
        $$.falselist = makelist(get_nextquad());
        emit (GOTO,NULL,NULL,NULL);}
         ;
```

A questo livello le espressioni comprendono gli operatori relazionali non inclusi precedentemente per la mancanza delle funzioni di gestione del backpatching.
Vengono generate due linee di codice per ogni operatore relazionale, una con il controllo della condizione di salto e il relativo salto se è vera, l'altra con il salto quando la condizione è falsa.
Chiaramente viene anche generato un temporaneo dove mantenere il valore dell' espressione.

```
smpl_expr   : term  { $$.temp = $1.temp; }
   | smpl_expr ADDOP mark term
       { if ($2 = OR)  {
       backpatch ($1.falselist,$3);
       $$.falselist = $4.falselist;
       $$.truelist                                    =
merge($1.truelist,$4.truelist);
       }
       else {
       $$.temp= newtemp (
             tipo_temporaneo($1.temp,$4.temp));
       emit($2,$$.temp,$1.temp,$4.temp); }
       }
       ;
```

La gestione del backpatching implica una differenziazione tra il caso degli operatori di somma e prodotto e quello dell' "or" logico.
Nei primi due casi, infatti, viene generata un'istruzione vera e propria, mentre con l'or devono solo essere opportunamente trattate gli attributi delle truelist e delle falselist.
Il marker serve per avere l'indirizzo dell'istruzione o delle istruzioni generate dal secondo termine dell' "or".

```
term     : factor           { $$.temp = $1.temp;
                            $$.truelist = $1.truelist;
                            $$.falselist = $1.falselist; }
       | term MULOP mark factor   { if ($2 = AND)
             {
             backpatch($1.truelist,$3);
             $$.truelist = $4.truelist;
             $$.falselist =
             merge($1.falselist,$4.falselist);
            }
             else {$$.temp= newtemp (
             tipo_temporaneo($1.temp,$4.temp));
             emit($2,$$.temp,$1.temp,$4.temp); }}
       ;
```

Anche in questo caso bisogna distinguere prodotto e divisione dall' "and" logico.
Il marker serve per tenere la posizione della seconda espressione.
Nel caso dell' "and" devo saltarvi se la prima è vera, cioè devo fare un controllo sia sulla prima che sulla seconda.
Al contrario, nel caso dell' "or" devo controllare la seconda espressione solo la prima è falsa.

```
factor  : ID                { $$.temp = lookup( $1 ) ; }
      | REALE               { $$.temp = const2cop(REALE, $1); }
      | INTERO              { $$.temp = const2cop(INTERO, $1); }
      | NOT factor          { $$.temp = $2.temp;
                            $$.truelist = $2.falselist;
                            $$.falselist = $2.truelist; }
      | '(' expression ')'     { $$.temp = $2.temp;
                            $$.truelist = $2.truelist;
                            $$.falselist = $2.falselist; }
      ;
```

Il not inverte la truelist con la falselist

```
mark    : { $$ = get_nextquad();}
mark_g  : { $$.truelist = makelist(get_nextquad());
            emit(GOTO,NULL,NULL,NULL);}
```

DEF.H

```
struct liste { int val;
               struct liste *next; } ;
```

Qui c'è la definizione del tipo LISTA, che è una lista di interi

```
typedef struct liste lista ;
```

```
typedef struct { int istr;
                 coppia *op1;
                 coppia *op2;
                 coppia *op3; } quadr;
```

Questo è il tipo che contiene la quadrupla *istruzione* più i *tre operandi*. Siccome bisogna gestire il backpatching, non è più possibile stampare a video il codice ma bisogna inserirlo in una struttura temporanea che mi consenta di effettuare le aggiunte successive.

```
/* coppia.h */
extern coppia *make_cop (int tipo, numero val);
extern numero get_val_cop(coppia *temp);
extern int get_tip_cop(coppia *temp);
extern coppia *const2cop(int tipo, numero n);
extern int tipo_temporaneo
(coppia *c1, coppia *c2);
extern int is_coppia_int(coppia *c);
```

Le funzioni che gestiscono gli oggetti di tipo coppia sono state spostate nel file coppia.c senza alcuna sostanziale modifica.

```
/* genera.h */
extern int get_istr_quad(quadr *q);
extern coppia *get_op1_quad(quadr *q);
extern coppia *get_op2_quad(quadr *q);
extern coppia *get_op3_quad(quadr *q);
extern void put_op1_quad(quadr *q, coppia *c);
extern void put_op2_quad(quadr *q, coppia *c);
extern void put_op3_quad(quadr *q, coppia *c);
extern quadr *get_quad_cod(int i) ;
extern int get_nextquad(void);
extern void emit(int tipo,coppia *t,coppia *x,coppia *y);
```

Sono state rielaborate le funzione di generazione del codice poichè ora non è più direttamente a video e incluse nuove funzioni per l'inserimento della quadrupla nella nuova struttura atta a contenere il codice.

```
/* stampa.c */
extern void print_symtbl(void);
extern void print_cod(void);
```

È stato creato un nuovo file con due funzioni per la stampa del codice e della symbol table.

```
/* liste.c */
extern int get_val_lista(lista *l);
extern lista *get_next_lista (lista *l);
extern lista *makelist( int val );
extern lista *addlist(lista *l,int n);
extern lista *merge(lista *l1, lista *l2 );

/* back.c */
extern void backpatch(lista *l, int quad);
```

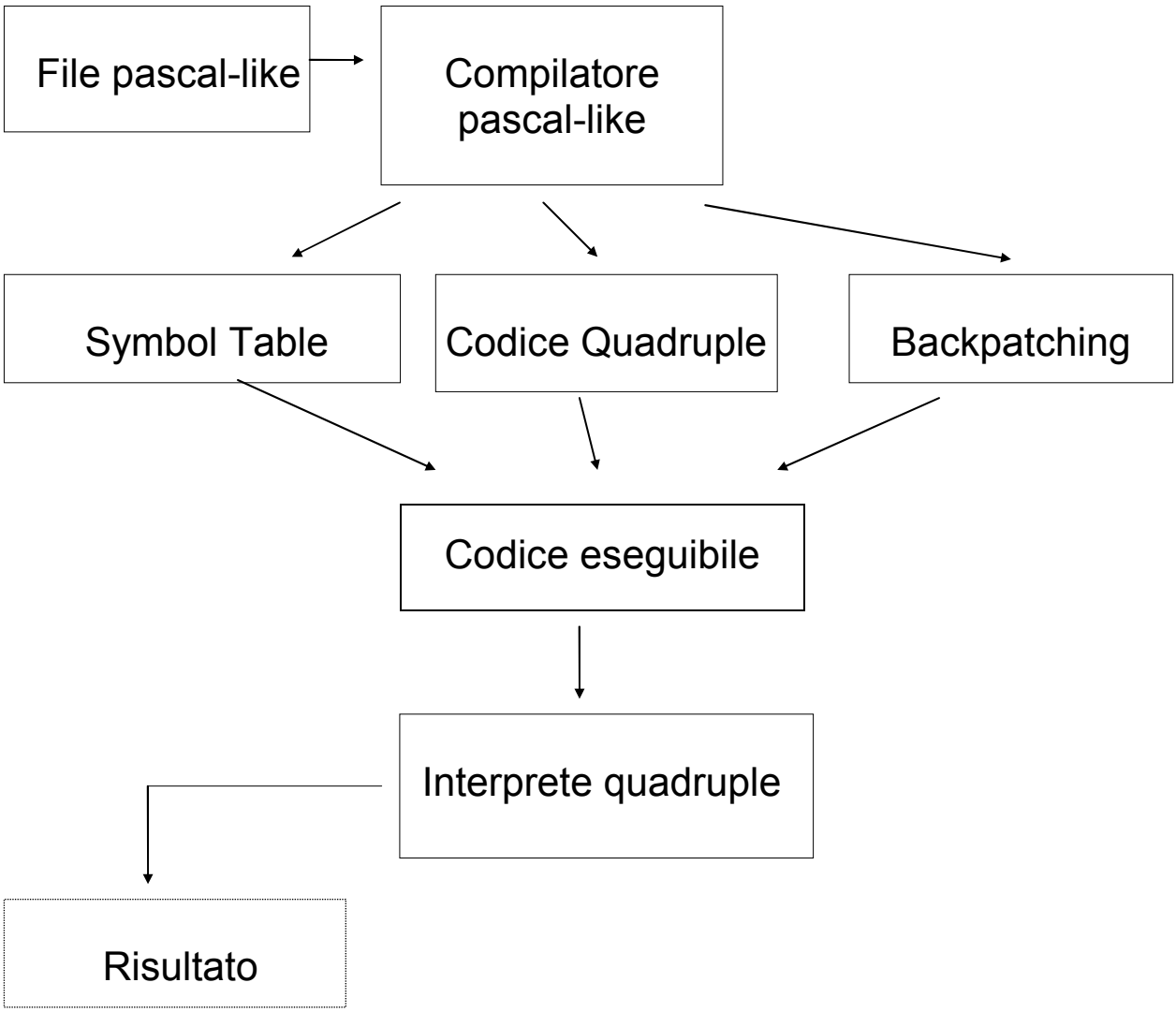
È stato creato un nuovo modulo anche per la gestione delle liste, indispensabile struttura per il backpatching.

È la funzione di backpatching vera e propria

LIVELLO 4

Con l'introduzione delle procedure il progetto del compilatore è completo. È stato necessario apportare alcune modifiche per poter inserire in symbol table i nuovi tipi di oggetto, cioè le procedure; per questo è stata definita una nuova costante intera *proc* oltre i precedenti *intero*, *reale* e *vartemp,* che viene utilizzata nel campo *tipo* della struttura coppia e della struttura *rec* (che contiene un elemento della symbol table).

ARCHITETTURA GENERALE DEL COMPILATORE



COMPONENTI DEL COMPILATORE

Il progetto del compilatore è composto da 6 moduli C, una specifica sintattica ed una lessicale.
I moduli C si chiamano rispettivamente:

SYM_TBL.C dove vengono descritte le funzioni di gestione della Symbol Table.

LISTA.C dove vengono descritte le funzioni relative alle liste usate per il backpatching.

BACK.C dove si trova la gestione vera e propria del backpatching, che si appoggia sulle procedure definite in "lista.c"

COPPIA.C al cui interno vi è il codice che riguarda la manipolazione di una delle strutture fondamentali, che serve per contenere gli operandi, siano essi costanti, variabili o procedure.

GENERA.C è il modulo di generazione del codice intermedio sotto forma di quadruple.

STAMPA.C contiene le funzioni di stampa della symbol table, del codice e delle liste.

Abbiamo, inoltre, l' header DEF.H per le definizioni fondamentali di costanti e dichiarazioni delle procedure esterne.
Infine, vi sono la specifica lessicale EXPR.L e la specifica sintattica SOURCE.Y

SPECIFICA SINTATTICA

```
%union {
        struct { coppia *temp1;
                 coppia *temp2; } variabile;
        int ty;
        int quadrup ;
        struct { int ty;
                 int dim; } tipo;
        struct {lista *list;
                 int param; } head; }
 /* parole chiave riservate */
%token  PROCEDURE
 /* operatori */
%token <op> LEGGI
%token <op> SCRIVI
%type <head> subprogram_head
%type <op> arguments
%type <op> parameter_list
%type <op> identifier_list
%type <op> declarations
/* Marker */
%type <quadrup> mark
%type <head> mark_g
```

```
%type <head> mark_a
 %%
Start : PROGRAM ID ';' mark_a declarations mark_g subprogram_declarations mark
compound_statement '.'
          { backpatch( $4.list, get_offset() );
            backpatch( $6.list, $8  );
            emit(HALT,NULL,NULL,NULL); }
            ;
subprogram_declarations : subprogram_declarations subprogram_declaration ';' { }
                                | { }
                                ;
subprogram_declaration : subprogram_head declarations compound_statement {
                backpatch($1.list, get_offset());
                emit(RETURN,const2cop(INTERO,$1.param),NULL,NULL);
                dec_livello(); }
                ;
subprogram_head : PROCEDURE ID arguments ';' {
                $$.list = makelist(get_nextquad());
                $$.param = $3;
                insert( $2, PROC );
                insert_param($3, $2) ;
                inc_livello();
                emit(ALLOC,NULL,NULL,NULL); }
                ;
arguments : '(' parameter_list ')' {$$=$2;}
                | {$$=0;}
                ;
parameter_list : identifier_list ':' type  { $$=$1;
                metti_tipo_var( $3.ty, $3.dim, $1); }
                | parameter_list ';' identifier_list ':' type { $$=$1+$3;
                metti_tipo_var( $5.ty, $5.dim, $3); }
                ;
statement : variable ASSIGNOP expression {
                if ($1.temp2 != NULL) $2 = XASSIGN;
                emit ($2, $1.temp1, $1.temp2, $3.temp); }
              | procedure_statement { }
              | compound_statement  { }
              | SCRIVI '(' expression ')' { emit (WRITE, $3.temp, NULL, NULL); }
              | LEGGI '(' expression ')' { emit (READ, $3.temp, NULL, NULL); }
              ;
procedure_statement : ID { emit(CALL, lookup($1), NULL, NULL); }
                            | ID '(' expression_list ')' { emit(CALL, lookup($1), NULL,
                            NULL); }
```

```
                    ;
expression_list : expression  { emit(PARAM,$1.temp, NULL,NULL);}
                | expression_list ',' expression { emit(PARAM, $3.temp, NULL,
                NULL );}
                ;
mark_a  : { $$.list = makelist(get_nextquad());
          emit(ALLOC,NULL,NULL,NULL); }
```

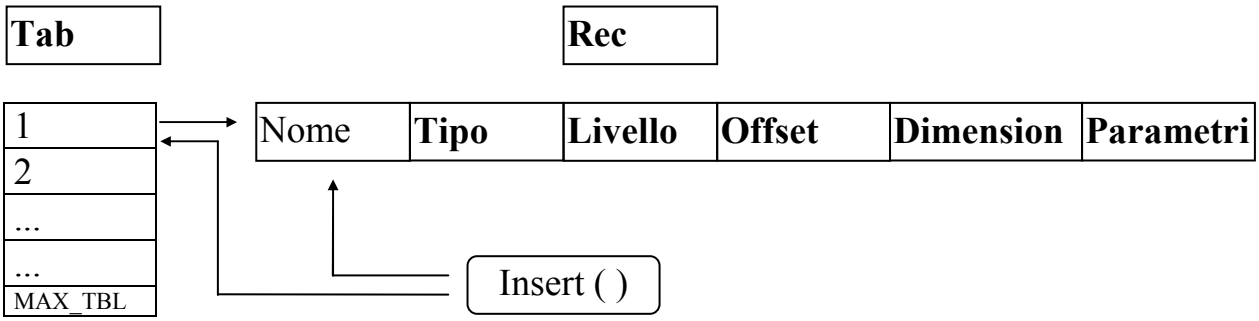
IDENTIFICATORI E COSTANTI

La specifica lessicale riconosce identificatori e costanti restituendo un intero relativo alla parola riconosciuta ed eventualmente un ulteriore attributo in questi sei casi:
ID (identificatore) - il nome dell' attributo.
INTERO - il valore intero
REALE - il valore reale
Quando viene riconosciuta una costante o un identificatore viene creata una struttura chiamata *coppia* (gestita in coppia.c) formata dai seguenti due campi: TIPO e VALORE.
Nel caso della costante il tipo può essere INTERO o REALE e il valore rappresenta il valore della costante.
Nel caso dell'identificatore il tipo è VARTEMP (infatti serve sia per variabili che per temporanei) e il valore, in questo caso, rappresenta la posizione in symbol table della variabile o del temporaneo, dove vengono inseriti tramite la funzione INSERT.

PROCEDURE

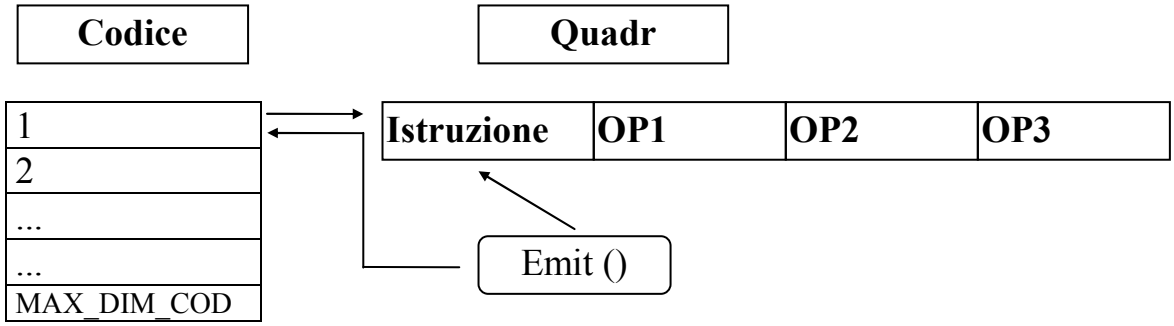
Anche le procedure vengono trattate in modo simile alle variabili e alle costanti; anche in questo caso, infatti, viene creata la coppia da passare come attributo nella specifica sintattica e viene inserito nella symbol table il nome della procedura insieme ai suoi dati con la medesima funzione INSERT.

SYMBOL TABLE



È un vettore di puntatori ad oggetti di tipo REC, tipo che è stato definito come struttura con sei campi: il primo è il *nome* della variabile, del temporaneo o della procedura; il secondo è il *tipo*, che mi permette di distinguere una variabile da una procedura (vartemp o proc); abbiamo poi il *livello* in cui è stato definito il simbolo; la *dimensione*, che serve nel caso degli array; l'*offset*, che, nel caso di variabili, contiene l'offset in memoria della variabile, oppure, nel caso di procedure, l'indirizzo di partenza della procedura; infine il *numero di parametri*, che rappresenta il numero di parametri della procedura e viene impostato a *no param* (-1) nel caso di variabili

MEMORIA



La memoria è formata da un vettore di quadruple, per cui è stato necessario definire una struttura atta a contenere i quattro campi della quadrupla, cioè ISTRUZIONE e i tre OPERANDI.
Questa struttura si chiama QUADR, il suo primo campo è un intero, mentre gli altri tre sono delle coppie, che, come abbiamo visto, possono contenere sia variabili che costanti.
Le funzioni che gestiscono la memoria sono definite in GENERA.C
La funzione principale è la EMIT, che, oltre ad inserire in memoria l'istruzione, converte il tipo degli operandi da intero a reale (generando l'istruzione ITOR) quando necessario.

BACKPATCHING

Per la gestione del backpatching è stato necessario definire delle funzioni ausiliarie per la gestione delle liste descritte nel file LISTE.C.

Vi sono quindi le funzioni:

MAKELIST	per la creazione di una lista
MERGE	per la fusione di due liste
ADDLIST	per l'aggiunta di un elemento alla lista

La funzione di backpatching principale si chiama "BACKPATCHING" e ha due parametri d'ingresso: il primo è una lista di interi, il secondo è un intero.
Ogni elemento della lista corrisponde a un indirizzo di memoria; viene quindi prelevata l'istruzione ad esso corrispondente e il secondo parametro viene inserito nel primo operando non istanziato dell'istruzione.

ATTRIBUTI

Sono stati definiti i seguenti attributi:

DVAL	contiene il numero intero o reale
EXPR	contiene nel primo campo un temporaneo, una variabile o una
costante	e negli altri due una lista per la true e la false list.
OP	contiene il tipo dell'operando
TIPO	contiene tipo e dimensione della variabile o del vettore
LISTE	contiene la true e la false list
HEAD	contiene una lista e il numero dei parametri della procedura