Progetto di linguaggi e compilatori

Traduttore MCL-C

(e script di compilazione GNU MCL via GCC)

Marco Arrigoni – 46522

Marco Balduzzi – 46525

Marco Gandolfi – 46524

Indice generale

1. Introduzione

Il progetto non ha la pretesa di introdurre un elemento innovativo (ne tanto meno utile) nel panorama dei linguaggi di programmazione. Abbiamo voluto sviluppare un linguaggio che, prendendo spunto da linguaggi diversi, ci permettesse di affrontare alcune problematiche peculiari di questi ultimi per poter fare pratica con la progettazione di un compilatore.

1.1 Finalità del linguaggio

Il Marco-Compact-Language (MCL) è stato sviluppato per fornire uno strumento semplice per la realizzazione di programmi di calcolo algoritmico e matematico in generale. E' possibile scrivere espressioni matematiche e gestirle attraverso condizioni di complessità crescente.

Come nella maggior parte dei linguaggi di programmazione è possibile fare uso di funzioni, cicli, espressioni condizionali e variabili locali e globali.

1.2 Caratteristiche del linguaggio

Il linguaggio è stato definito Compact perché utilizza una sintassi compatta, lineare e funzionale.

1.2.1 Variabili

Le variabili hanno tutte tipo in virgola mobile e non necessitano di dichiarazione. L'inizializzazione di una variabile può avvenire in uno dei due modi seguenti:

```
a

b = 3
```

Nel primo caso viene dichiarata una variabile `a` cui viene assegnato il valore 0, mentre nel secondo caso si dichiara la variabile `b` a cui viene direttamente assegnato il valore dell'espressione a destra del segno di uguaglianza (in questo caso 3).

Gli identificatori delle variabili sono stringhe che non possono iniziare con cifre. Possono contenere, oltre a caratteri dell'alfabeto anglosassone e cifre, il simbolo underscore (_).

1.2.2 Input

È possibile richiedere l'interazione dell'utente per assegnare valori alle variabili a tempo di esecuzione dallo standard input. La sintassi è molto compatta e consiste nel far seguire il simbolo dollaro (`$`) al nome della variabile a cui si vuole assegnare il valore. Questa direttiva non deve essere seguita da `;`

```
x $
```

1.2.3 Output

La stampa a video viene fatta evitando di terminare l'istruzione con un punto e virgola (questa caratteristica è stata presa dal linguaggio di scripting di Matlab). Le istruzioni dell'esempio precedente stampano entrambe sullo standard output il valore della variabile inizializzata (0 la prima e 3 la seconda). Per evitare che vengano stampati in output i valori delle variabili è necessario scriverle come segue:

```
a;
b = 3;
```

È inoltre possibile stampare a video delle stringhe di testo semplicemente inserendole tra apici:

```
'Stringa di esempio...'
```

1.2.4 Commenti

I commenti sono preceduti dal simbolo di cancelletto (`#`), si estendono fino al termine della linea e non vengono in nessun caso stampati a video. All'interno un commento può

essere costituito da una sequenza alfanumerica qualsiasi.

1.2.5 Operatori

Sono definite le operazioni matematiche di base (somma, sottrazione, prodotto, quoziente) utilizzando le classiche norme di precedenza degli operatori (prodotto e quoziente hanno la precedenza rispetto a somma e sottrazione) e la notazione infissa. Ovviamente è possibile fare uso di parentesi tonde annidate per definire espressioni complesse.

```
z*(3.14+y)/((z-5)/4)
```

Inoltre è possibile utilizzare gli operatori condizionali < (minore), > (maggiore), = (uguale) e logici `&&` (and), `||` (or) e `!` (not). Anche nelle espressioni logiche è concesso l'uso di parentesi, se non utilizzate la precedenza degli operatori è quella standard (not precede or che precede and).

```
((a || d) && !b) || c || (b = c)
```

1.2.6 Blocchi condizionali e iterativi

Il linguaggio supporta la gestione del classico blocco condizionale if-fi e di quello iterativo while-elihw.

```
if(cond)
      # istruzioni ...
fi


while(cond)
      # istruzioni ...
elihw
```

In entrambi i casi il controllo della condizione viene effettuato all'ingresso del blocco. Nel primo caso il codice nel blocco viene eseguito una volta sola, nel secondo caso il codice viene eseguito ripetutamente finché è vera la condizione `cond` (`cond` > 0).

1.2.7 Funzioni

Tutte le funzioni sono racchiuse tra i blocchi `begin` (a cui segue il nome della funzione e la lista dei parametri racchiusa tra parentesi tonde) e `end`. La procedura principale, che viene eseguita all'avvio del programma, deve essere identificata con il nome `program` e può essere definita con un numero illimitato di argomenti letti da linea di comando. Questa ha quindi forma del tipo:

```
begin program(x, y, z)
      #istruzioni ...
end
```

La funzione principale non restituisce nessun dato. È possibile definire funzioni che restituiscano un dato di tipo in virgola mobile, anteponendo al nome della funzione nella dichiarazione, un nome di variabile locale in cui deve essere memorizzato il valore da restituire. Per esempio si potrebbe definire una semplice funzione che calcola una potenza data una base e un esponente intero, nel modo seguente:

```
begin ris = potenza(base, esponente)
      ris = 1;
      while(esponente>0)
            ris = ris * base;
            esponente = esponente - 1;
      elihw
end
```

La chiamata delle funzioni di supporto avviene secondo una sintassi standard. La

funzione `potenza(base,esponente)` dell'esempio precedente può essere chiamata all'interno delle espressioni come segue:

```
c = potenza(a,b) * 5.125  # assegnamento


if(potenza(x,y) > z)  # confronto
      #istruzioni ...
fi
```

1.2.8 Istruzione di uscita

L'esecuzione del codice può essere interrotta in qualsiasi punto del programma inserendo l'istruzione `exit` (che può essere seguita o meno da `;`).

1.2.9 Un esempio

Ecco un esempio per riassumere le caratteristiche del linguaggio:

```
# dichiarazione di una variabile globale a;


# dichiarazione di una funzione # NB: deve essere fatta prima di esse-
re usata
begin q = quad(n)
  q = n*n;
end


# funzione principale (entry point) # riceve due parametri d ed e
dalla linea di comando
begin program(d, e)
  # l'istruzione seguente assegna ad a il valore 5 # e lo stampa a vi-
  deo (perché non c'è il ;) a = 3+2
```

```
  # stampa il valore ricevuto dalla linea di comando
  e
  # legge una variabile dallo standard input
  'Inserire il valore di x:'
  x$
  # d è riconosciuta valida perché letta da linea di # comando
  a = d+1+x;

  # assegna a b il quadrato di a (uso della funzione # quad)
  b = quad(a);

  # blocco condizionale if() if (b<c)
  # l'istruzione seguente deve generare un # errore in quanto la va-
  riabile s non è stata # definita
  a = s;
  else
  # vado a dichiarare e inizializzare una nuova s s = a+b;
  # esempio di stampa di una stringa "Fine" # non c'è il punto e
  virgola 'Fine'
  fi

  # blocco iterativo while() while(! (b = 0))
  if(b>0)
  # riduce s e ne stampa il valore b = b-1
  else
  # chiude il programma se b<0 exit;
  fi
  elihw
end
```

1.3 Obiettivo del progetto

Vogliamo scrivere un traduttore che, dato in ingresso del codice sorgente scritto secondo le specifiche MCL, ne analizzi la correttezza e lo traduca in un sorgente C che, compilato, permetta di eseguire le operazioni matematiche indicate. Questo doppio passaggio di compilazione ci permette di evitare di avere a che fare con il linguaggio macchina di una specifica architettura (permettendoci ad esempio di cross-compilare i programmi C generati dal compilatore MCL su architetture diverse da quella per cui il compilatore è stato scritto) e sopratutto di focalizzare l'attenzione sulle particolarità che contraddistinguono l'MCL e demandare al compilatore C alcuni controlli tipici sulle variabili e sulle funzioni.

Uno script in Bash si occuperà di gestire le due fasi del processo di compilazione in modo automatico.

Il traduttore è in grado di verificare la correttezza sintattica e semantica di un codice sorgente MCL. Inoltre sono implementati controlli relativi a funzioni e variabili per far fronte alle differenze tra la sintassi dell'MCL e quella del C.

1.4 Tecnologia utilizzata

Nella prima fase di implementazione abbiamo individuato e valutato i diversi software disponibili per lo sviluppo del traduttore. La comunità del Free Software mette a disposizione gratuitamente numerose soluzioni che permettono di definire scanner e parser utilizzando i più comuni linguaggi di programmazione. La nostra scelta è caduta sugli strumenti Flex e Bison, che nel corso degli anni sono diventati lo standard per la scrittura di compilatori e traduttori in ambiente Free Software e GNU/Linux.

Lo scanner verrà generato con Flex, acronimo di Fast Lexical Analyzer (il cui sito Internet di riferimento è http://flex.sourceforge.net). Si tratta di una versione open-source di Lex un generatore di analizzatori lessicali originariamente sviluppato dalla AT&T e diventato lo standard su sistemi Unix, in quanto facente parte dello standard POSIX.

Bison è il generatore di parser GNU, il cui progetto è liberamente disponibile sul sito GNU http://www.gnu.org/software/bison/. Anche Bison è la versione open-source di un software sviluppato da AT&T per i sistemi Unix, Yacc (yet another compiler compiler).

1.5 Milestones

10/04/2006: completamento dello scanner.

29/05/2006: completamento del parser.

15/06/2006: consegna del progetto completo di documentazione.

Nota bene: per un report di lavoro più dettagliato si veda il diary file (Changelog) riportato in capitolo 9.

2. Grammatica BNF

2.1 Alfabeto

L'alfabeto del linguaggio MCL è costituito dai seguenti terminali:

Σ ={COMMENTO, FINELINEA, TEXT, PUNTOVIRGOLA, MBEGIN, END, TEXT, UGUALE, PARENTESI_AP, PARENTESI_CH, PROGRAM, VIRGOLA, EXIT, FRASE, IF, FI, ELSE, WHILE, ELIHW, OPMATE1, OPMATE2, NUMERO, OR, AND, NOT, OPLOGIC, UGUALE}

Terminale	Significato / Valore di ritorno
COMMENTO	Restituisce il commento → #bla bla bla
FINELINEA	Identifica il '\n'. Non restituisce nulla
TEXT	Identificatori per le variabili
PUNTOVIRGOLA	Restituisce il carattere punto virgola ;
MBEGIN	Restituisce la parola chiave `begin`
END	Restituisce la parola chiave `end`
PARENTESI_AP	Restituisce il carattere parentesi aperta (
PARENTESI_CH	Restituisce il carattere parentesi chiusa)
PROGRAM	Restituisce la parola chiave `program`
VIRGOLA	Restituisce il carattere virgola ,
EXIT	Restituisce la parola chiave `exit`
FRASE	Restituisce una frase, identificata da un apice all'inizio e un apice alla fine
IF	Restituisce la parola chiave `if` (usata per aprire un blocco `if`)
FI	Restituisce la parola chiave `fi` (usata per chiudere un blocco `if`)
ELSE	Restituisce la parola chiave `else` (usata per definire un'alternativa alla condizione del blocco `if`)

Terminale	Significato / Valore di ritorno
`WHILE`	Restituisce la parola chiave `while` (usata per aprire un blocco `while`)
`ELIHW`	Restituisce la parola chiave `elihw` (usata per chiudere un blocco `while`)
`OPMATE1`	Restituisce un operatore `+` o `-`
`OPMATE2`	Restituisce un operatore `*` o `/`
`NUMERO`	Identifica i numeri (con o senza virgola)
`OR`	Restituisce il carattere `||` (identifica un'operazione di OR)
`AND`	Restituisce il carattere `&&` (identifica un'operazione di AND)
`NOT`	Restituisce il carattere `!` (che identifica la negazione)
`OPLOGIC`	Restituisce un operatore di confronto `<` o `>`
`UGUALE`	Restituisce il carattere uguale `=`
`GET`	Restituisce il carattere `$` (usato per leggere una variabile dallo standard input)

2.2 Significato dei non-terminali

Non-terminale	**Descrizione**
`S`	Assioma
`FINE_L`	Identifica il non terminale per il fine linea
`CL`	Comment List: lista dei commenti
`VL`	Variable List: lista delle variabili globali
`PL`	Function List: lista delle funzioni
`M`	Main
`FL`	Function: definizione della funzione

Non-terminale	**Descrizione**
`FUNCT`	Id della funzione (nome e lista dei parametri)
`BODY`	Insieme delle istruzioni
`RET`	Identifica la variabile di ritorno di una funzione o di una espressione (RET = TEXT UGUALE)
`PARAM_L`	Lista dei parametri
`PARAM_L2`	Seconda lista dei parametri (contiene i parametri dal secondo all'ultimo)
`INSTR`	Non-terminale dell'istruzione
`PV`	Punto virgola
`CALL_FUNCT`	Non-terminale per una chiamata a funzione
`ESP`	Espressione
`COND`	Condizione (utilizzato nei cicli `if` e `while`)
`CALL_PARAM`	Lista dei parametri di una chiamata di funzione
`CALL_PARAM2`	Seconda lista dei parametri di una chiamata di funzione (contiene i parametri dal secondo all'ultimo)
`T`	Utilizzato nelle espressioni, indica il secondo termine dell'espressione (il primo è ancora un'espressione)
`F`	Utilizzato nelle espressioni con operazioni di moltiplicazione e divisione, identifica il secondo fattore
`A`	Utilizzato nelle condizioni, identifica la seconda “variabile” utilizzata nella condizione
`C`	Utilizzato nelle condizioni con AND, identifica la seconda “variabile” utilizzata nella condizione
`PRED`	Predicato in una condizione con operatori di confronto
`OPC`	Non terminale che identifica il tipo di controllo in esame

2.3 Grammatica

Analizziamo ora la struttura dell'albero sintattico utilizzando la Bacus Naur Form partendo dall'assioma:

```
S   →   FINE_L CL VL PL M

FINE_L   →    ϵ

FINE_L   →   FINELINEA FINE_L
```

Questa produzione definisce i vari blocchi che compongono un programma scritto in MCL. Il primo non terminale (FINE_L) rappresenta la possibilità di inserire, prima del codice, una serie di spazi bianchi. Gli altri non terminali identificano, rispettivamente, la lista dei commenti, delle variabili globali, delle funzioni definite e il main del programma

```
CL   →    ϵ

CL   →   COMMENTO FINE_L CL
```

Questo gruppo rappresenta la lista dei commenti. Un commento può stare su una sola riga e più commenti possono essere separati anche da una serie di righe.

```
VL   →    ϵ

VL   →   TEXT PUNTOVIRGOLA FINE_L CL VL
```

Per rappresentare invece la lista delle variabili globali vengono usate queste due produzioni. Viene data la possibilità di inserire una serie di commenti (CL) tra una variabile globale e l'altra

```
PL   →    ϵ

PL   →   PL FL CL
```

In MCL c'è la possibilità di definire delle funzioni prima della definizione del main. E'

inoltre previsto l'inserimento di una serie di commenti dopo la funzione, per spiegarne il funzionamento.

```
FL   →   MBEGIN RET FUNCT BODY END FINE_L

FL   →   MBEGIN FUNCT BODY END FINE_L
```

Come già visto, le funzioni definibili possono essere di due tipi: quelle che restituiscono un valore e quelle che non restituiscono un valore. Le prime sono identificate dalla prima produzione vista la presenza del non-terminale RET mentre le seconde, dall'altra produzione.

```
RET   →   TEXT UGUALE
```

Questa produzione viene utilizzata per definire la variabile di ritorno nelle funzioni che restituiscono un valore.

```
FUNCT   →   TEXT PARENTESI_AP PARAM_L PARENTESI_CH
```

Per definire il nome e i parametri della funzione principale, viene definito questo non-terminale.

```
M   →   MBEGIN PROGRAM PARENTESI_AP PARAM_L PARENTESI_CH BODY END
FINE_L
```

Il main, per essere valido, deve prevedere la struttura definita da questa produzione.

```
PARAM_L   →    ϵ

PARAM_L   →   TEXT PARAM_L2

PARAM_L2   →    ϵ

PARAM_L2   →   VIRGOLA TEXT PARAM_L2
```

Per definire la lista dei parametri vengono utilizzate queste 4 produzioni.

- caso 1: non ci siano parametri: l'unica produzione ridotta sarà la prima.
- caso 2: un solo parametro: le produzioni coinvolte sono la seconda e la terza. In pratica viene identificata il parametro e la produzione poi si riduce a ϵ
- caso 3: più di un parametro: le produzioni coinvolte sono la seconda (per il primo parametro), la terza (per ridurre la produzione 2) e la quarta (per tutti gli altri parametri)

```
BODY  →   ϵ

BODY  →  FINE_L INSTR FINE_L BODY
```

In questo caso viene gestito il blocco delle istruzioni. Si prevede che una istruzione possa essere preceduta e seguita da una serie di righe vuote (identificate da FINE_L). In più una istruzione (tralasciando le righe vuote) sarà necessariamente seguita da un'altra istruzione. Nel caso in cui non ci siano più istruzioni, la produzione BODY viene ridotta grazie alla presenza di ϵ .

```
PV  →   ϵ

PV  →  PUNTOVIRGOLA
```

Come spiegato nel paragrafo 1.2.2, nel nostro linguaggio è prevista la possibilità di mettere o meno il punto virgola alla fine delle istruzioni. Questa capacità viene implementata grazie a queste due produzioni.

```
INSTR  →  EXIT PV

INSTR  →  FRASE

INSTR  →  COMMENTO

INSTR  →  CALL_FUNCT PV

INSTR  →  RET ESP PV

INSTR  →  ESP PV

INSTR  →  TEXT GET
```

```
INSTR  →  IF COND BODY FI

INSTR  →  IF COND BODY ELSE BODY FI

INSTR  →  WHILE COND BODY ELIHW
```

Tutte queste produzioni rappresentano tutte le possibili istruzioni scrivibili in MCL. Le prime 3 sono rispettivamente: la possibilità di scrivere l'istruzione exit (per terminare il programma), di scrivere una frase (da stampare in output) o di scrivere un commento all'interno del codice del programma. Viene poi riconosciuta la possibilità di effettuare chiamate a funzione e di scrivere delle espressioni (che possono ritornare dei valori o no). Segue l'istruzione per leggere input dallo standard input. Infine vengono riconosciuti i blocchi di condizione if e if-else, assieme alla possibilità di definire cicli while (precisamente le ultime 3 produzioni).

```
CALL_FUNCT  →  TEXT PARENTESI_AP CALL_PARAM PARENTESI_CH
```

Per effettuare una chiamata a funzione, è necessario inserire una parte di testo (che identifica il nome della funzione richiamata) e gli parametri che devono essere passati alla funzione.

```
CALL_PARAM  →  ϵ

CALL_PARAM  →  ESP CALL_PARAM2

CALL_PARAM2  →  ϵ

CALL_PARAM2  →  VIRGOLA ESP CALL_PARAM2
```

Per definire la lista degli parametri di una chiamata di funzione, viene usata la stessa logica vista nella lista dei parametri (di una definizione di funzione). L'unica differenza è che, in questo caso, è prevista la possibilità di inserire delle espressioni, come valori da passare. Espressioni che, come vedremo più avanti, potranno essere espressioni matematiche, chiamate ad altre funzioni o anche semplici variabili.

```
ESP  →  ESP OPMATE1 T
```

```
ESP  →  OPMATE1 T

ESP  →  T
```

Queste tre produzioni vengono utilizzate per definire espressioni. Più precisamente, la prima definisce espressioni di somma e sottrazione (opmate1 è un terminale che può assumere i valori + o -), la seconda per espressioni con segno e la terza per definire altri tipi di espressione oppure per aggiungere alle espressioni dei valori numerici, di variabile o i risultati di una chiamata a funzione

```
T  →  T OPMATE2 F

T  →  F
```

In questo caso viene definita l'espressione di moltiplicazione o di divisione (opmate2 è il terminale che può assumere i valori * o /).

```
F  →  CALL_FUNCT

F  →  NUMERO

F  →  TEXT

F  →  PARENTESI_AP ESP PARENTESI_CH
```

Questa produzione serve per identificare i possibili elementi di una espressione: chiamate a funzione, numeri, variabili o anche altre espressioni, delimitate da parentesi tonde.

```
COND  →  PARENTESI_AP COND OR A PARENTESI_CH

COND  →  A
```

Per definire le condizioni (per un if o per un while) vengono utilizzate queste produzioni. Per imporre la precedenza dell'operatore OR rispetto all'operatore AND, è stata sviluppata una struttura a due livelli. La prima è una condizione con l'operatore OR:

```
A   →   PARENTESI_AP A AND C PARENTESI_CH

A   →   C
```

In questo caso la condizione espressa è con l'operatore AND:

```
C   →   COND

C   →   PARENTESI_AP NOT COND PARENTESI_CH

C   →   PARENTESI_AP PRED PARENTESI_CH

C   →   F
```

Gli elementi che possono essere contenuti in una condizione. Quindi: altre condizioni, condizioni negate o predicati (questi ultimi due delimitati da parentesi tonde). Nelle condizioni, possono esserci anche numeri, variabili o anche chiamate a funzioni (C → F, dove questa F è la stessa vista precedentemente nelle espressioni).

```
PRED   →   ESP OPC ESP
```

Un predicato è invece dato dal confronto tra due espressioni. Il confronto viene fatto utilizzando gli operatori di >, < o =.

3. Analizzatore lessicale (scanner)

Flex è uno strumento che permette di generare degli scanner. Uno scanner è un programma che riconosce i pattern lessicali in un testo. Flex legge file scritti secondo il linguaggio di definizione di Lex. Flex genera come output un file sorgente C, “lex.yy.c” per default, il quale definisce una routine `yylex()`. Questo file può essere compilato e linkato con la libreria di runtime di Flex per produrre un eseguibile. Quando il programma viene eseguito, analizza il suo input alla ricerca delle espressioni regolari. Quando ne trova una esegue il corrispondente codice C.

3.1 Scrittura dello scanner

La struttura di un file di definizione in formato Lex/Flex (che ha estensione .l) è piuttosto semplice. Si devono creare tre sezioni separate dal marcatore `%%`:

```
definizioni
%%
regole
%%
codice
```

3.1.1 Definizioni

Nella prima sezione, oltre ad eventuali direttive per il generatore Flex, è possibile definire gli header da inserire nel codice C dello scanner generato racchiusi tra i simboli `%{` e `%}`.

Nel nostro caso questa parte si presenta come segue:

```
%{
#include <string.h>
#include <stdio.h>
#include "mcl.tab.h"
u_short num_linea=1;
%}
```

Oltre all'inclusione delle librerie per la gestione delle stringhe e dell'I/O lo scanner necessita dell'inclusione del file header che definisce la classe del parser che sarà poi generata da Bison (in modo da creare il collegamento tra scanner e parser).

Abbiamo inoltre definito la variabile globale `num_linea` per poter tener traccia del numero di linea che il compilatore analizza in un determinato punto. Questo sarà utile in fase di sviluppo e debugging del compilatore e dei programmi MCL.

In seguito è possibile definire degli identificatori per le espressioni regolari più comuni. Questo permette di riutilizzare la stessa espressione regolare senza doverla riscrivere ogni volta. Nel nostro caso abbiamo voluto utilizzare questa funzionalità per identificare due gruppi generici di caratteri. Il primo costituito da tutte le lettere dell'alfabeto anglosassone (sia minuscole che maiuscole), il secondo costituito dalle cifre da 0 a 9:

```
NUMERO      [0-9]

NOME        [A-Za-z_]
```

3.1.2 Regole

La descrizione delle regole è costituita da coppie di pattern lessicali e azioni, dove il pattern può essere un espressione regolare, un set di caratteri determinato, oppure una definizione scritta nella sezione precedente e l'azione è un frammento di codice C eseguito ogni volta che nel testo di input viene rilevato il pattern definito a sinistra.

Se vi sono più regole che corrispondono a una certa stringa di testo, Flex esegue l'azione associata alla stringa più lunga.

Quando viene individuato un pattern, il valore della stringa viene memorizzato all'interno della variabile `yytext` . Il token viene restituito con il comando `return nome_token`, dove `nome_token` deve essere definito nell'header generato da Bison (come descritto nel prossimo capitolo). L'oggetto token può essere strutturato a piacere in modo da memorizzare attributi diversi a seconda del token.

Di seguito è riportato la sezione delle regole definite per il nostro scanner.

```
      /* keywords */
begin return MBEGIN;
end   return END;
if    return IF;
fi    return FI;
else  return ELSE;
while return WHILE;
elihw return ELIHW;
program     return PROGRAM;

      /**** operatori matematici ****/
("+"|"-")   {strcpy (yylval.stringa, yytext); return OPMATE1;}

("*"|"/")   {strcpy (yylval.stringa, yytext); return OPMATE2;}

      /**** operatori logici ****/
(">"|"<")   {strcpy (yylval.stringa, yytext); return OPLOGIC;}
"!"   return NOT;

"="   return UGUALE;

      /**** usato per gestire l'input ******/
"$"   return GET;

      /**** parentesi ****/
"("   return PARENTESI_AP;
")"   return PARENTESI_CH;
```

```
      /**** virgola...separa attributi/parametri ****/
","   return VIRGOLA;

"||"  return OR;

"&&"  return AND;

exit  return EXIT;

\n    {num_linea++; return FINELINEA; }
;     return PUNTOVIRGOLA;

      /**** identificatore numeri ****/
{NUMERO}+("."{NUMERO}+)?     {strcpy (yylval.stringa, yytext); return
NUMERO;}

      /**** identificatore variabili ****/
{NOME}({NOME}|{NUMERO})*     {strcpy (yylval.stringa, yytext); return
TEXT;}

      /**** commenti ****/
"#"[^\n]*         {strcpy (yylval.stringa, yytext); return COMMENTO;}

      /**** frasi ****/
"'"[^\']*\' {strcpy (yylval.stringa, yytext); return FRASE;}

<<EOF>>     yyterminate();

" "   ;     // eliminiamo lo spazio
```

```
\t    ;      //eliminiamo il tab


.     {
            printf ("*** Scanner: carattere non riconosciuto alla linea
%d: %s\n",num_linea, yytext);
            yyterminate();
      }
```

L'espressione regolare `{NUMERO}+("."{NUMERO}+)?` effettua il match su numeri (interi o con virgola) e quella `{NOME}({NOME}|{NUMERO})*` sulle variabili. Le variabili devono iniziare con un carattere testuale e possono poi contenere numeri o caratteri a piacimento.

L'espressione regolare che identifica i commenti è composta da due parti. Il carattere # e una qualsiasi sequenza di caratteri che non diversi dal carattere di fine linea. Mentre l'espressione regolare che effettua il match delle frasi (stringhe di testo da riprodurre in output) è composta da tre parti: il primo carattere di apice, una sequenza di caratteri qualsiasi escluso l'apice e l'apice di chiusura.

La maggior parte delle azioni restituisce semplicemente il token. Questo avviene nei casi in cui il pattern è costituito da una sequenza definita di caratteri (come avviene per le parole chiave). Per i pattern che possono restituire stringhe diverse (operatori logici e matematici, numeri, variabili, commenti e stringhe di testo) è necessario memorizzare il valore della stringa riconosciuta nell'attributo `yylval.stringa` definito nel parser in modo che possa essere utilizzato da quest'ultimo.

Quando viene rilevato il carattere di fine linea, oltre produrre il corrispondente token, viene incrementato il contatore `num_linea`.

La regola `<<EOF>> yyterminate()` indica di terminare la scansione una volta raggiunta la fine del file di input. Le regole con azioni vuote (per spazi e tabulazioni) indicano di non eseguire nessuna azione su questi pattern. In questo modo il compilatore ignorerà la spaziatura all'interno del codice.

L'ultima regola riconosce invece qualsiasi tipo di pattern non definito in precedenza.

L'azione associata genera un messaggio di errore indicando il numero dell'ultima linea letta e termina la scansione.

3.2 Verifica dello scanner

Prima dell'implementazione del parser ci siamo preoccupati di verificare la correttezza dello scanner. Sono stati fatti molti test sulla capacità di interpretazione sintattica di stringhe di input di varia natura.

3.2.1 Scanner di test

Parallelamente alla versione vista sopra, abbiamo sviluppato una versione standalone dello scanner ridefinendo le regole, scrivendo cioè a schermo il tipo di token riconosciuto invece di ritornarlo al parser (tutte le regole del tipo `return nome_token` sono state sostituite da qualcosa come `printf("nome_token\n");`). Sonp stati tolti tutti i riferimenti agli header generati da Bison ed inserita una funzione `main` da eseguire all'avvio del programma. In questa funzione si indica a Flex di leggere dallo standard input (`yyin = stdin;`) e di invocare la funzione `yylex()` che si occupa di effettuare la scansione dell'input consumando i token ed eseguendo le azioni associate.

Di seguito è riportato il codice dello scanner usato per i test:

```
%{
#include <string.h>
#include <stdio.h>
u_short num_linea=1;
%}
NUMERO      [0-9]
NOME  [A-Za-z_]
%%
begin printf("MBEGIN\n");
end   printf("END\n");
```

```
if    printf("IF\n");
fi    printf("FI\n");
else  printf("ELSE\n");
while printf("WHILE\n");
elihw printf("ELIHW\n");
program     printf("PROGRAM\n");

("+"|"-")   {printf("OPMATE1 %s\n", yytext);}
("*"|"/")   {printf("OPMATE2 %s\n", yytext);}

(">"|"<")   {printf("OPLOGIC %s\n", yytext);}
"!"   printf("NOT\n");

"="   printf("UGUALE\n");

"$"   printf("GET\n");

"("   printf("PARENTESI_AP\n");
")"   printf("PARENTESI_CH\n");

","   printf("VIRGOLA\n");

"||"  printf("OR\n");

"&&"  printf("AND\n");

exit  printf("EXIT\n");
```

```
\n    {num_linea++; printf("FINELINEA %s\n", yytext); }

;     printf("PUNTOVIRGOLA\n");

{NUMERO}+("."{NUMERO}+)?      {printf("NUMERO %.2f\n",atof(yytext));}

{NOME}({NOME}|{NUMERO})*      {printf("TEXT %s\n", yytext);}

"#"[^\n]*   {num_linea++; printf("COMMENTO %s\n", yytext);}

"'"[^\']*\' {printf("FRASE %s\n",yytext);}

<<EOF>>     yyterminate();

" "   ;     // eliminiamo lo spazio
"     "     ;     //eliminiamo il tab

.     {
            printf ("*** Scanner: carattere non riconosciuto alla linea
%d: %s\n",num_linea, yytext);
            yyterminate();
      }

%%

main(int argc, char **argv)
{
// per il test leggiamo da stdin
    yyin = stdin;
```

```
// run dell'analizzatore lessicale
    yylex();
}
```

3.2.2 Esempio di sessione di debugging dello scanner

Compiliamo lo scanner con flex e gcc, ricordandoci di linkarlo con le liberie flex e andando a produrre il binario di test mcl-scanner:

```
$ flex mcl-test.l
$ gcc lex.yy.c -o mcl-scanner -lfl
```

Mandando in esecuzione lo scanner in questo modo l'input viene catturato da linea di comando:

```
$ ./mcl-scanner
#questa e' una prova dello scanner (commento)
COMMENTO #questa e' una prova dello scanner (commento)
FINELINEA
```

Questo sarà l'entry point del programma:

```
begin program() # main
MBEGIN
PROGRAM
PARENTESI_AP
PARENTESI_CH
COMMENTO # main
FINELINEA
```

Verifica di espressioni condizionali e matematiche:

```
(3<2)+a;

PARENTESI_AP

NUMERO 3.00

OPLOGIC <

NUMERO 2.00

PARENTESI_CH

OPMATE1 +

TEXT a

PUNTOVIRGOLA

FINELINEA
```

Assegnamento con espressione matematica e richiamo alla funzione quad():

```
a=b+2*quad(c);

TEXT a

UGUALE

TEXT b

OPMATE1 +

NUMERO 2.00

OPMATE2 *

TEXT quad

PARENTESI_AP

TEXT c

PARENTESI_CH

PUNTOVIRGOLA

FINELINEA
```

Prova dei singoli terminali:

```
||
OR
FINELINEA

&&
AND
FINELINEA

*
OPMATE2 *
FINELINEA

/
OPMATE2 /
FINELINEA

+
OPMATE1 +
FINELINEA

-
OPMATE1 -
FINELINEA

!
NOT
FINELINEA

=
```

```
UGUALE

FINELINEA


<
OPLOGIC <

FINELINEA


>
OPLOGIC >

FINELINEA
```

Gli errori sintattici vengono correttamente riconosciuti:

```
ciao.

TEXT: ciao

*** Scanner: carattere non riconosciuto alla linea 1: .



baccalà

TEXT: baccal

*** Scanner: carattere non riconosciuto alla linea 1:
```

(il nostro è uno scanner ASCII)

4. Analizzatore sintattico (parser)

Bison è un generatore di parser general-purpose che converte una descrizione di una grammatica LALR(1) in un programma C che effettua il parsing di quella grammatica. Bison è compatibile con Yacc: tutte le grammatiche scritte correttamente in Yacc devono funzionare con Bison senza cambiamenti.

La sintassi di Bison è essenzialmente una versione machine-readable della “Bacus-Naur Form” (BNF).

Il parser MCL è stato scritto nel linguaggio di Bison e integrato con lo scanner, modificato a sua volta per garantire un corretto flusso di informazioni tra lo scanner e il parser. L’accoppiamento dei due strumenti fornisce un framework molto efficiente che ci ha permesso di implementare il traduttore MCL-C non con troppe difficoltà. Il funzionamento complessivo del traduttore è molto semplice: lo scanner riconosce i singoli token (begin, if, end, testo..) e li comunica al parser insieme al relativo valore. E’ poi compito del parser di riconoscere la struttura del linguaggio e di fare gli opportuni controlli semantici. Infine traduce il sorgente nel corrispondente in linguaggio C; è poi lo GNU C Compiler (gcc) che compila il sorgente C e genera il codice eseguibile.

La catena completa è quindi

SORGENTE MCL

ANALISI LESSICALE: lo scanner esegue l'analisi lessicale

ANALISI SINTATTICA: lo scanner esegue i controlli dell'analisi sintattica

TRADUZIONE IN C: lo scanner esegue la traduzione in linguaggio C

COMPILAZIONE: infine gcc compila il sorgente C

BINARIO

Lato utente è fornito lo script gmcl (GNU MCL compiler) per l'analisi, traduzione e compilazione di sorgenti scritti secondo le specifiche del linguaggio MCL.

4.1 Traduttore

La procedura di traduzione in linguaggio C necessita di leggere ed interpretare tutte le istruzioni del codice sorgente per rappresentarle globalmente in memoria. Tale rappresentazione risulta necessaria poiché non è possibile eseguire una traduzione on-the-fly.

Per questo motivo si è deciso di implementare un albero semantico, in cui la radice corrisponde all'assioma della grammatica BNF. L'albero è formato da tre campi:

- dat: campo di informazione. E' di tipo char e contiene le informazioni riguardo il nodo corrente (cioè se nel nodo in esame rappresenta una istruzione piuttosto che una definizione di una funzione, o quant'altro)
- sin / des: sono puntatori ai nodi successivi (sinistra e destra). Questo ci ha permesso di suddividere l'albero semantico in "zone", ognuna delle quali rappresenta una parte del programma da tradurre.

Questa struttura dati ci ha permesso di dividere il programma in varie parti e in modo più intuitivo perchè, per esempio l'assioma è costituito sostanzialmente da due parti: una parte *variabili globali* e una parte *funzioni (funzioni e main)*. Quindi l'albero semantico fatto in questo modo è risultato molto utile poiche per la radice, a sinistra si è messa la lista delle variabili globali, con i commenti, mentre a destra la lista delle funzioni e il main (che può essere considerato come una funzione)

(qui di seguito è riportato, come esempio di quanto detto, la creazione della radice per l'assioma)

```
S → FINE_L CL VL PL M
```

```
head = mknodo("inizio",
                    mknodo("globals",(nodo*)$2,(nodo*)$3),
                    mknodo("function",(nodo*)$4,(nodo*) $5));
```

Nel parser abbiamo potuto definire i non-terminali come se fossero dei nodi dell'albero. Questo ci ha portato dei grossi vantaggi in quanto ci ha risolto un problema, che si andrà a spiegare di seguito.

Problema:
Le regole semantiche di una produzione vengono eseguite solo al momento della riduzione della produzione. Quindi vengono creati prima i nodi foglia e poi i nodi a cui attaccare queste "foglie". Il problema è: dove attacco queste foglie se non ho ancora un nodo?

Il problema è appunto stato risolto dichiarando i non-terminali di tipo nodo. In questo modo quando una produzione viene ridotta, l'azione semantica crea un nodo sul non-terminale e questo poi viene passato alla produzione da cui si proviene (e via così fino alla riduzione dell'assioma).

Esempio di creazione del nodo commento:

```
CL → COMMENTO FINE_L CL

$$ = (void *) mknodo("commento", mknodo(buf,NULL,NULL), (nodo *) $3);
```

Con `'$$'` in pratica, si definisce che CL (il non-terminale della produzione) è un nodo dell'albero semantico che viene passato alla produzione da cui proviene.

4.2 Controlli semantici

Nel parser sono stati implementati anche tre controlli semantici. Questi sono: controllo sulla dichiarazione delle variabili, controllo sulla dichiarazione delle funzioni e controllo sul numero di parametri in una chiamata di funzione. Per poter ottenere ciò è stata introdotta una ulteriore struttura dati: una lista di liste.

4.2.1 La lista di liste

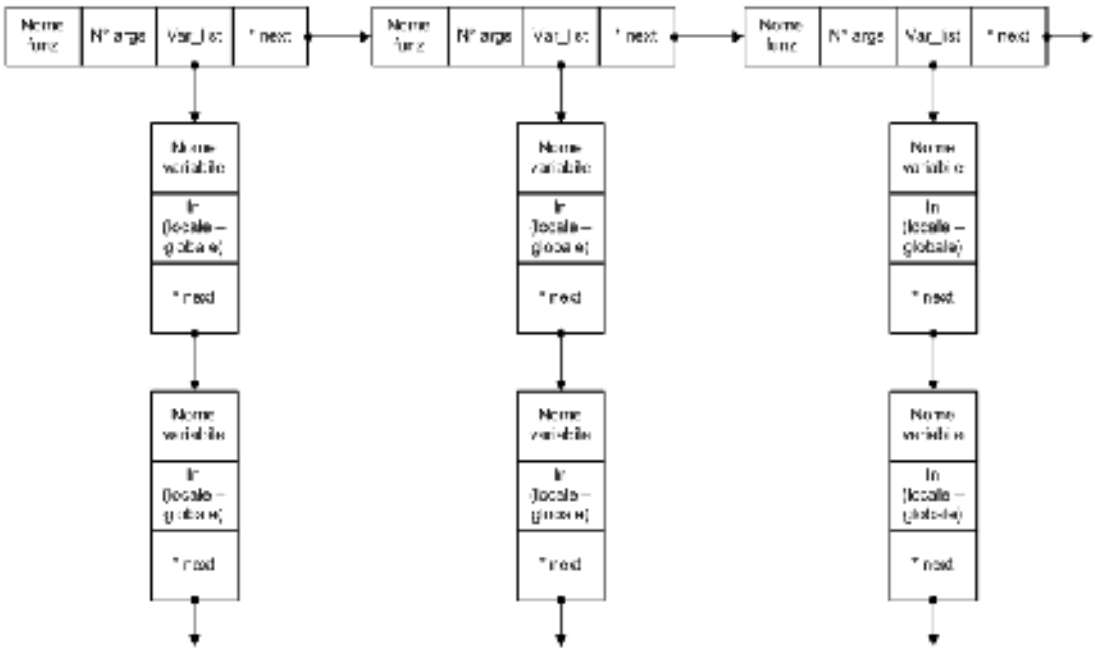
Come dice il nome, la lista di liste è composta da due liste: la prima rappresenta una successione di nodi contenenti informazioni riguardanti le funzioni, mentre la seconda coniente informazioni sul contenuto delle funzioni.

In particolare, la lista *funct_list* (lista delle funzioni) contiene 4 campi:

- funct_name: nome della funzione
- n_args: numero di argomenti della funzione
- var_list: puntatore alla lista delle variabili locali
- next: puntatore al nodo successivo

mentre la seconda lista, *var_list* contiene 3 campi:

- var: nome della variabile
- in: identifica se una variabile è stata definita locale oppure fa parte della lista dei parametri della funzione
- next: puntatore al nodo successivo della lista



Le due liste hanno una gestione diversa per quanto riguarda l'inserimento dei nodi. La

lista delle funzioni ha un inserimento in coda (ogni nuovo nodo viene inserito in fondo alla lista), mentre per la lista delle variabili, ogni nuova variabile viene inserita in testa alla lista.

Il salvataggio di queste informazioni consente di definire controlli sulla dichiarazione delle variabile e sulle funzioni.

Di seguito verranno spiegati i controlli semantici implementati. Per quanto riguarda le funzioni che li implementano fare riferimento al paragrafo 4.3 .

4.2.2 Controllo sulla dichiarazione delle variabili

Uno dei controlli semantici effettuati dal compilatore, riguarda la dichiarazione delle variabili. Visto che in MCL le variabili non hanno una dichiarazione vera e propria, la cattura di tutte queste variabili non è stata così semplice.

Le variabili da “catturare” sono sostanzialmente in tre posizioni differenti:

- variabili globali
- variabili dei parametri delle funzioni
- variabili locali, interne al codice

Le prime due sono le più semplici da catturare. Le variabili globali, che si trovano all'inizio del codice MCL, vengono aggiunte alla lista di liste, nella funzione “255”. Si è deciso di adottare questo nome perchè la grammatica non prevede nomi di funzione di questo tipo (un nome di funzione deve necessariamente iniziare con una lettera) e quindi, in questo modo si possono evitare conflitti con funzioni realmente dichiarate.

Le variabili contenute nei parametri di una funzione vengono catturate in modo semplice, quando vengono ridotte le rispettive produzioni.

Per quanto riguarda le variabili locali, queste vengono catturate in queste 2 situazioni:

- `"a ="` (la variabile 'a' viene catturata e aggiunta alla lista delle variabili locali, se non è ancora stata aggiunta)
- quando si trova all'interno di una espressione

Le variabili vengono prima inserite in una lista temporanea e, al momento della creazione del nodo della funzione, viene accodata al nodo. Questo perchè la produzione che "genera" la funzione, viene ridotta soltanto dopo che è stata ridotto tutto il corpo della funzione.

4.2.2.1 Esempio di verifica duplicazione variabili

I controlli semantici del parser danno errore nel momento in cui la stessa variabile è stata dichiarata due volte, come:

```
glob1;

glob1;
```

Ecco come si comporta il traduttore:

```
$ ./mcl < test.mcl

DEBUG ENABLED with -DMDEBUG

[cut]

ERROR: global variable glob1 already declared. Abort
```

4.2.3 Controllo sulla dichiarazione delle funzioni

Questo controllo viene eseguito in fase di compile-time. In pratica, durante la generazione dell'albero semantico, ogni volta che viene ridotta una produzione di una funzione, questa viene inserita nell'albero solo se non è già stata dichiarata. L'inserimento della funzione viene fatto utilizzando la chiamata a funzione *push_funct*, mentre il controllo se esiste già, con la *check_funct* (queste due funzioni sono spiegate rispettivamente nei paragrafi 4.3.2 e 4.3.5).

4.2.3.1 Esempio di verifica duplicazione funzioni

I controlli semantici del parser danno errore nel momento in cui la stessa funzione è stata dichiarata due volte, come:

```
begin moltiplicazione (fattore1, fattore2);

begin moltiplicazione (fattore1, fattore2);
```

- Sorgente con funzioni duplicate

```
begin funct1 (a)
      'ciao sono la funzione1'
end

begin funct1 (a)
      'ciao sono la funzione1'
end

begin program()
      'test duplicazione funzioni'
end
```

Ecco come si comporta il traduttore:

```
$ ./mcl < test-funct.mcl
DEBUG ENABLED with -DMDEBUG
DEBUG: parser push_funct(): adding function 255 with #param = 0
DEBUG: parser check_funct(): checking _function_ 255 in the list (#
var = 0)
DEBUG: parser check_funct(): [KO] function not found!
DEBUG: parser push_funct(): [OK] added function 255
DEBUG: parser PARAM_L: added new _var_ a (1)
DEBUG: parser push_funct(): adding function funct1 with #param = 1
DEBUG: parser check_funct(): checking _function_ funct1 in the list (#
var = 1)
```

```
DEBUG: parser check_funct(): [KO] function not found!

DEBUG: parser push_funct(): [OK] added function funct1

DEBUG: parser PARAM_L: added new _var_ a (1)

DEBUG: parser push_funct(): adding function funct1 with #param = 1

DEBUG: parser check_funct(): checking _function_ funct1 in the list (#
var = 1)

DEBUG: parser check_funct(): [OK] function found

DEBUG: parser push_funct(): [KO] function funct1 already present ...
```

ERROR: function funct1 already declared. Abort!

4.2.4 Controllo sul numero di parametri nella chiamata di funzioni

Il compilatore effettua anche un controllo sul numero di parametri immessi in una chiamata di funzione, per verificare che sia coerente con la dichiarazione.

Questa operazione viene fatta in due fasi: la prima in cui si contano i parametri della funzione dichiarata e la seconda in cui si contano i parametri della chiamata di funzione.

La prima fase, viene fatta in fase di compilazione, quando viene creato l'albero semantico. In pratica, ogni volta che viene ridotta una produzione di `PARAM_L` e di `PARAM_L2` viene incrementato un contatore `param_counter`. Questo contatore verrà poi salvato nella lista di liste al momento dell'inserimento della funzione nella lista. (Il contatore, una volta salvata l'informazione nella lista viene azzerato, in modo da essere pronto per contare i arametri della funzione successiva)

La seconda fase, invece, viene fatta in fase di traduzione. Quando il traduttore (che è ancora il compilatore) trova una chiamata di funzione da tradurre, vengono contati il numero di parametri da stampare e, se questo valore con coincide con il valore contenuto nella lista (per quella funzione), la traduzione si blocca, restituendo un errore

4.2.4.1 Esempio di verifica del numero di parametri delle funzioni

Nel seguente esempio la funzione moltiplica accetta come parametri i due fattori e ri-

torna il loro prodotto. Nell'esempio funzionante vengono passati correttamente due valori, mentre nel secondo il numero di parametri non è coerente con la dichiarazione.

- Sorgente funzionante. Da notare le caratteristiche “compact” dell'MCL, che garantiscono pulizia di codice (vedi la chiamata a funzione moltiplica() e relativa stampa del valore di ritorno) e compattezza (6 righe in mcl contro 11 righe in C)

```
begin prodotto = moltiplica (fattore1, fattore2)
        prodotto = fattore1*fattore2;
end
begin program()
      moltiplica(3,4)
end
```

- Corrispettivo sorgente in C:

```
float moltiplica (float fattore1, float fattore2)
{
  float prodotto;
  prodotto = fattore1 * fattore2;
  return (prodotto);
}
int main ()
{
  printf ("%.4f\n", moltiplica (3, 4));
  return 0;
}
```

- L'esecuzione stampa semplicemente il prodotto di 3 e 4

```
$ ./test-param
12.0000
```

- Sorgente errato: un solo parametro passato alla funzione moltiplica():

```
      begin program()
```

```
                moltiplica(3)

      end
```

```
ERROR: Number of parameters in function "moltiplica" exceed with defi-
nition

ERROR: Expecting  2 parameters instead of 1


Translation error
```

4.3 Funzioni e procedure implementate nel parser

Verranno ora spiegate le funzioni e le procedure implementate nel parser, necessarie per la compilazione e la traduzione del linguaggio MCL

4.3.1 La funzione mknodo

```
nodo *mknodo (char d[256], nodo *sx, nodo *dx)
{
  nodo *temp = (nodo*) malloc(sizeof(nodo));
  if (temp==NULL) {
            fprintf (stderr, "ERROR: malloc() on nodo\n");
            exit (1);
       }
  strncpy(temp->dat, d, strlen(d)*sizeof(char));
  temp->sin = sx;
  temp->des = dx;
  return (temp);
}
```

La funzione mknodo viene utilizzata per la creazione dell'albero durante la compilazione del linguaggio MCL. Come si può vedere dal codice sopra riportato, la funzione riceve come parametri una stringa (sottoforma di array di char) e due puntatori di tipo *nodo* (destra e sinistra). In pratica, viene creato un nodo dell'albero e, dopo essere stato inizializzato, viene restituito come risultato della funzione. Questo consente di passare il nodo tra le varie produzioni (una volta ridotte), fino a formare l'albero completo

4.3.2 La funzione push_funct

```
int push_funct(char dato[256],int n)
{
	...
	if (!check_funct(dato)) // non trovato
	{
		funct_prop	*new_node	=	(funct_prop*)	malloc(sizeof
(funct_prop));

		strcpy(new_node->funct_name, dato);
		new_node->n_args = n;

		if (strcmp(dato,"255") && tmp_var_list)
		{
			list *p = tmp_var_list;

			while (p!=NULL)
			{
				...
				//inserimento in testa
				...
			}
		}
		else
			new_node->var_list=NULL;

		new_node->next = NULL;

		if (funct_list == NULL) // la lista delle funz e' vuota
			funct_list = new_node;
		else		// ci sono già delle funzioni
		{
			funct_prop *p = funct_list;
			while (p->next != NULL) // vado alla fine alla fine
				p = p->next;
```

```
                  p->next = new_node;
            }
            ...
            return 1;   // ok
      }
      else
      {
#ifdef MDEBUG
      if (strcmp(dato,"255"))
            fprintf(stderr, "DEBUG: parser push_funct(): [KO] function
%s already present ...\n",dato);
#endif
            return 0;   // ko
      }
      return 0;
}
```

Questa funzione viene richiamata ogni volta che viene ridotta una produzione di una definizione di funzione. La funzione riceve in ingresso un identificativo della funzione che si sta aggiungendo e un intero che contiene il numero di parametri della funzione. Il primo controllo che viene fatto riguarda l'esistenza della funzione (attraverso la funzione *check_funct* spiegata più avanti). Se la funzione esiste già, viene riportato un messaggio d'errore e la funzione restituisce 0. Altrimenti viene aggiunta in coda alla lista principale. Una volta fatto ciò, viene aggiunta la lista delle variabili (con inserimento in testa) e viene riportato 1 (1 = ok; 0 = ko).

4.3.3 La funzione push_var

```
int push_var(char dato[256], char f[256])
{
      funct_prop *p = funct_list;
      list *t, *t_p;
      u_short funct_found=0;

      while (p!=NULL)
      {
            if (!strcmp(p->funct_name,f))
            {
                  funct_found=1;
                  t=p->var_list;
```

```
                  break;
            }
            p=p->next;
      }

      if (!funct_found)
            return 0;

      t_p=t;
      while (t_p!=NULL)
      {
            if (!strcmp(t_p->var, dato))
                  return 0;
            t_p=t_p->next;
      }
      list *new_node = (list *) malloc(sizeof(list));
      strcpy(new_node->var, dato);
      new_node->next = t;
      p->var_list=new_node;
      return 1;
}
```

Questa funzione viene utilizzata per inserire le variabili globali. La particolarità di questa funzione è che in questo caso, il nodo funzione esiste già (ed è in prima posizione) quindi l'inserimento viene fatto direttamente nella lista di liste. La prima parte del corpo della funzione riguarda la ricerca della funzione “255” all'interno della lista, mentre la seconda, l'inserimento delle varie variabili globali nella sottolista *var_list*

4.3.4 La funzione push_global_var

```
int push_global_var(char dato[256], u_short in)
{
      // vediamo se e' gia' in lista
      if (tmp_var_list!=NULL) {
            list *p=tmp_var_list;
            while (p!=NULL) {
                  if (!strcmp(p->var,dato))
                        return 0;
                  p=p->next;
```

```
            }
      }
/* Gestione doppione variabili globali/locali */
      if (in == 0)
      {
            list *temp = funct_list->var_list;
            while (temp!= NULL)
            {
                  if (!strcmp(temp->var,dato))
                        return 0;
                  temp=temp->next;
            }
      }
/********************/
      // ok inseriamo
      list *new_node = (list *) malloc(sizeof(list));
      strcpy(new_node->var, dato);
      new_node->in=in;
      new_node->next = tmp_var_list;
      tmp_var_list=new_node;
      return 1;   // ok
}
```

Questa funzione viene invece utilizzata per inserire nella lista di liste tutte le variabili locali e i parametri delle funzioni. All'inizio viene verificato se la variabile esiste già. Se non esiste viene verificata l'esistenza nelle variabili globali (solo se la variabile non fa parte dei parametri della funzione) e, se anche questo controllo porta esito negativo, allora viene aggiunta in testa alla lista delle variabili.

4.3.5 La funzione check_funct

```
int check_funct(char dato[256])
{
      funct_prop *p=funct_list;

      while (p != NULL)
      {
```

```
            if (!strcmp(p->funct_name, dato))
                  return 1;
            else
                  p = p->next;
      }
      return 0;
}
```

Questa è una delle funzioni più importanti. Viene infatti utilizzata per verificare se una funzione è già stata inserita all'interno della lista di liste. Se la funzione esiste già viene restituito 1 mentre se non esiste viene restituito 0.

4.3.6 La funzione check_var

```
int check_var(char dato[256], char f[256])
{
      funct_prop *p=funct_list;
      int funct_found = 0;

      /*
            prima vediamo se e' globale!
      */
      list *t = funct_list[0].var_list;

      // variabili globali
      while (t!=NULL)
      {
            if (!strcmp(t->var, dato))
                  return 1;

            t=t->next;
      }

      /* prendiamo la funzione che ci interessa e la scorriamo */
      while (p!=NULL)
      {
            if (!strcmp(p->funct_name,f))
```

```
                {
                        funct_found=1;
                        break;
                }
        else
                p=p->next;
    }

    if (!funct_found)
        return 0;

    // variabli della funzione
    t = p->var_list;
    while (t!=NULL)
    {
        if (!strcmp(t->var, dato))
            return 1;

        t=t->next;
    }
    return 0;
}
```

Simile alla funzione precedente, questa verifica se una variabile esiste o meno. Il primo controllo viene fatto nelle variabili globali, il secondo invece all'interno della funzione, tra le variabili locali già aggiunte. Come prima, se esiste già una variabile con lo stesso nome, questa funzione restituisce 1, altrimenti 0.

4.3.7 Le funzione print_xxx

Esistono poi una serie di funzioni utilizzate per stampare a video la traduzione dal linguaggio MCL a C. La logica di funzionamento è, bene o male, la stessa per tutte: viene analizzato il campo informazione di ogni nodo dell'albero e, in base al suo contenuto, viene richiamata l'apposita funzione *print* per effettuare la stampa a video. La particolarità di queste funzioni è la ricorsività della chiamata: per esempio se trovo una espressione, verrà chiamata in modo ricorsivo la stampa degli elementi dell'espressione,

scorrendo l'albero. Una volta terminata la stampa dell'espressione, si torna alla funzione di stampa delle istruzioni, si passa al nodo successivo e, sempre ricorsivamente, viene richiamata la *print_instr* (stampa istruzione) per gestire il nuovo nodo. L'utilizzo di questa tecnica è risultato molto vantaggioso, perchè in modo semplice e immediato, si è riusciti a gestire in modo efficiente l'albero e lo scorrimento dello stesso.

5. Script di supporto lato utente

All'utente vengono forniti i due script:

- build-mcl
- gmcl

5.1 build-mcl per la compilazione del traduttore (installazione)

Lo script build-mcl deve essere eseguito la prima volta e consente di compilare il traduttore MCL-C generando l'eseguibile **mcl**

Tale script esegue le seguenti operazioni.

./build-mcl: esecuzione dello script di compilazione

eliminazione di vecchie versioni e file temporanei

bison -d mcl.y: generazione del parser

flex mcl.y: generazione dello scanner

gcc: generazione del traduttore

mcl: traduttore

5.2 gmcl: il compilatore GNU MCL

La compilazione di sorgenti scritti in linguaggio MCL è resa semplice attraverso l'uso dello script di supporto gmcl. Tale script traduce il sorgente in linguaggio C, lo indenta e lo compila, generando l'eseguibile.

gmcl accetta in ingresso come parametro il nome del sorgente MCL senza estensione.

Per esempio la compilazione di un programma di calcolo matriciale **matrix.mcl** è articolata nel seguente modo:

./gmcl matrix

./mcl < matrix.mcl > matrix.c: traduzione in sorgente c

indent matrix,c: indentazione del codice c

gcc matrix.c -o matrix: generazione dell'eseguibile matrix.c

5.3 Esempio di installazione del traduttore e compilazione di un sorgente MCL (da linea di comando)

Il traduttore MCL-C viene fornito in codice aperto.

Il pacchetto contiene:

- i due script lato utente build-mcl e gmcl
- il sorgente flex mcl.l
- il sorgente bison mcl.y

Al momento del primo utilizzo è necessario compilare i sorgenti con lo script build-mcl, in questo modo:

```
$ ./build-mcl
MCL-C Translator builder

* Cleaning old files
* Compiling the parser mcl.y with bison
```

```
* Compiling the scanner mcl.l with flex
* Compiling mcl

Done! Compiler gmcl ready :)

Run compiler with ./gmcl <filename>
                  <filename> should be without .mcl extension...
```

Per compilare un sorgente MCL si consiglia l'utilizzo dello script di alto livello gmcl, che traduce e compila il sorgente automaticamente.

Vogliamo compilare il sorgente test1.c :

```
$ ./gmcl test1
MCL-C Translator

* Translating
* Indenting code
* Compiling
Done!   Binary file is test1
        Source C file is test1.c (indented)
```

Altrimenti è possibile tradurre il sorgente manualmente con il comando

./mcl < sorgente.mcl > sorgente.c

preoccupandosi poi di compilare il sorgente c.

6. Funzionalità rilevanti

6.1 Gestione input

La gestione dell'input a runtime è stata implementata sia a livello di scanner che di parser. Nel dettaglio lo scanner riconosce il carattere dollaro ($) come token GET. Quindi il parser è in grado di riconoscere le istruzioni di input nella forma variabile$. La corrispondente BNF è:

INSTR: TEXT GET

La corrispondente traduzione in C è realizzata utilizzando la funzione scanf("%s", &nome_variabile), in questo modo:

```
else if (!strcmp(p->dat,"get"))
{
        printf("scanf(\"%%f\", ");
        printf("&%s",p->sin->dat);
        printf(");\n");
}
```

6.1.1 Esempio di programma che calcola il prodotto di due numeri letti a runtime

- Sorgente MCL

```
# programma che calcola a*b
begin program()
        'a='
        a$
        'b='
        b$

        'il prodotto vale'
```

```
        a*b
end
```

- Sorgente C

```
int
main ()
{
// function variables
  float a;
  float b;
  puts ("a=");
  scanf ("%f", &a);
  puts ("b=");
  scanf ("%f", &b);
  puts ("il prodotto vale");
  printf ("%.2f\n", (float) a * b);

  return 0;
}
```

- Esecuzione

```
$ ./test_input
a=
2
b=
5
il prodotto vale
10.00
```

6.2 Gestione input da linea di comando

La gestione dell'input da linea di comando è realizzata semplicemente con una azione semantica che traduce gli argomenti della procedura program() con le dichiarazioni float var = (atof) argv[i], ovvero:

```
// print dei parametri
if (p->sin==NULL)
   printf ("\nint main ()\n{\n");
  else
  {
     printf ("\nint main (int argc, char **argv)\n{\n");
     nodo *temp = p;
     while(temp->sin != NULL)
       {
          temp = (nodo*) temp->sin;
          printf("%s = atof (argv[%d]);\n", temp->dat, i);
          i++;
       }
  }
```

Ciò che viene fatto è il controllo dell'esistenza di parametri e il recupero di tali parametri dalla lista temp->sin, per poi convertirli in una istruzione di assegnazione in linguaggio C.

6.2.1 Esempio di programma che calcola la potenza di due numeri da linea di comando

- Sorgente MCL

```
begin q=mclpow(b,e)
```

```
      #init
      i=0;
      q=1;

      while (i<e)
            q=q*b;
            i=i+1;
      elihw
end

# prende due numeri da linea di comando e li eleva a potenza
begin program (a,b)
      'questo programma fa a^b'
      a
      '^'
      b
      '='
      c=mclpow(a,b)
end
```

- Sorgente C

```
// Generated by MCL-C Translator
//--------------------

#include <stdlib.h>
#include <stdio.h>

// function variables
```

```
float
mclpow (float b, float e)
{
// function variables
  float q;
  float i;
// init
  i = 0;
  q = 1;

  while (i < e)
    {
      q = q * b;
      i = i + 1;

    }
  return (q);
}

//  prende due numeri da linea di comando e li eleva a potenza

int
main (int argc, char **argv)
{
  float a = atof (argv[1]);
  float b = atof (argv[2]);
// function variables
  float c;
```

```
  puts ("questo programma fa a^b");
  printf ("%.2f\n", (float) a);
  puts ("^");
  printf ("%.2f\n", (float) b);
  puts ("=");
  c = mclpow (a, b);
  printf ("\n  c =\n      %.2f\n\n", c);

  return 0;
}
```

- Esecuzione

```
$ ./a.out 2 6
questo programma fa a^b
2.00
^
6.00
=

  c =
      64.00
```

7. Interfaccia grafica

Per facilitare le operazioni principali di traduzione e compilazione di un programma scritto con il linguaggio MCL, abbiamo sviluppato un'interfaccia grafica utilizzando Kommander un potente tool messo a disposizione dall'ambiente grafico KDE.

7.1 Kommander

Kommander permette di creare rapidamente finestre di dialogo facilmente integrabili con l'architettura di KDE attraverso il protocollo DCOP. È costituito da due parti. La prima è un editor grafico in cui è possibile sviluppare le finestre di dialogo e le applicazioni ed editare gli elementi di scripting. La seconda parte è l'esecutore che processa il file XML generato e manda in esecuzione l'interfaccia grafica. Kommander non compila applicazioni ma non costruisce la sua interfaccia attraverso scripting interpretato e le sue chiamate DCOP sono funzioni compilate. In questo modo produce finestre di dialogo e applicazioni leggere e al tempo stesso veloci.

7.2 Funzionalità

L'interfaccia prevede tre aree: a destra l'area di testo in cui viene caricato il codice MCL, a sinistra in alto c'è lo spazio per il codice C e in basso la console in cui viene visualizzato l'output delle operazioni eseguite (figura 1).

È possibile caricare file con estensione mcl nell'area di testo apposita oppure scrivere direttamente il codice. Questa funzionalità permette di produrre rapidamente dei piccoli programmi in formato MCL. Per caricare un file si inserisce la posizione del file sorgente MCL nella casella di testo in alto. Il pulsante a lato apre una finestra di dialogo per navigare all'interno del file system e scegliere il file (figura 2).

Una volta aperto il file, il contenuto viene caricato nell'area di testo a sinistra (figura 3). Dopo il caricamento è ancora possibile effettuare modifiche al codice. L'operazione di traduzione (che si avvia facendo click su pulsante “Traduci”) infatti avvia la traduzione mandando direttamente il testo contenuto nell'area di testo al traduttore MCL.

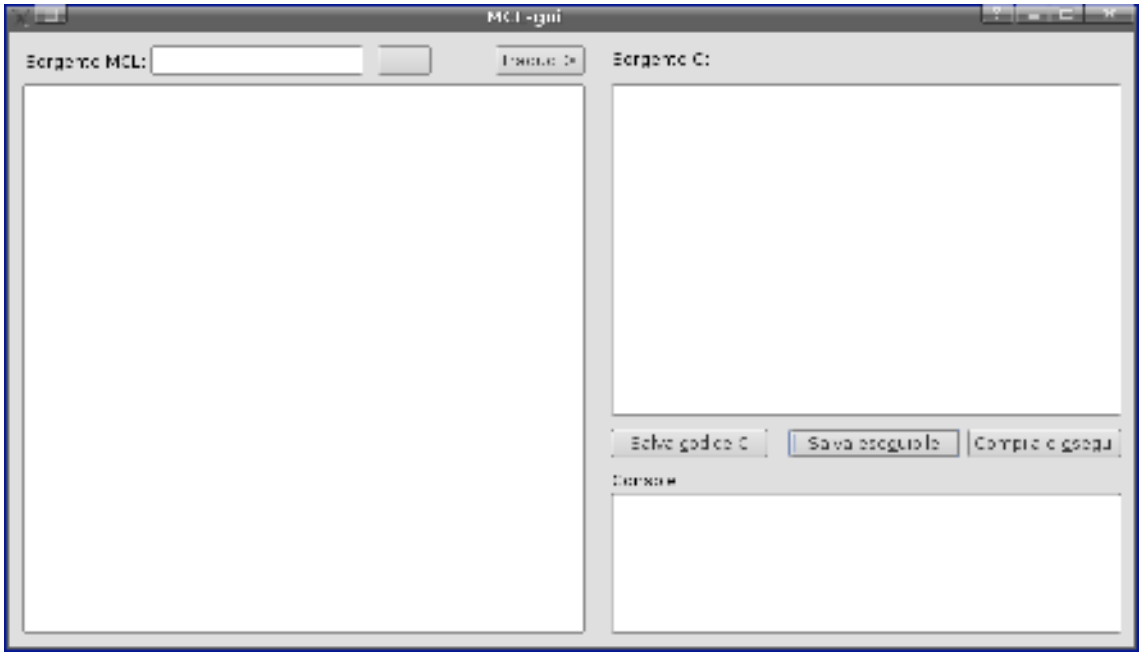


Illustrazione 1: Interfaccia grafica

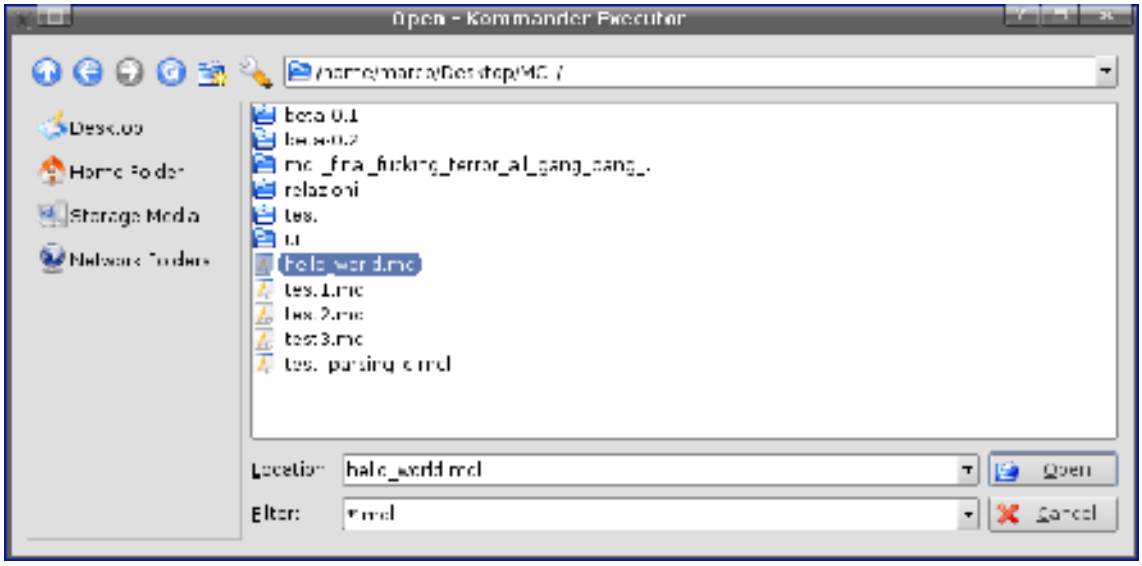


Illustrazione 2: Finestra di dialogo per l'apertura del file

Il risultato della traduzione viene indentato e inserito nell'area di testo la quale è liberamente editabile qualora si volessero apportare modifiche al codice C (figura 4). Qualora la traduzione non venga portata a termine correttamente, perché ad esempio il traduttore MCL incontra errori sintattici nel codice di input, la console riporta un errore di traduzione, mentre la descrizione dell'errore, con i messaggi generati dallo scanner o dal parser, viene data nell'area di testo (figura 5).

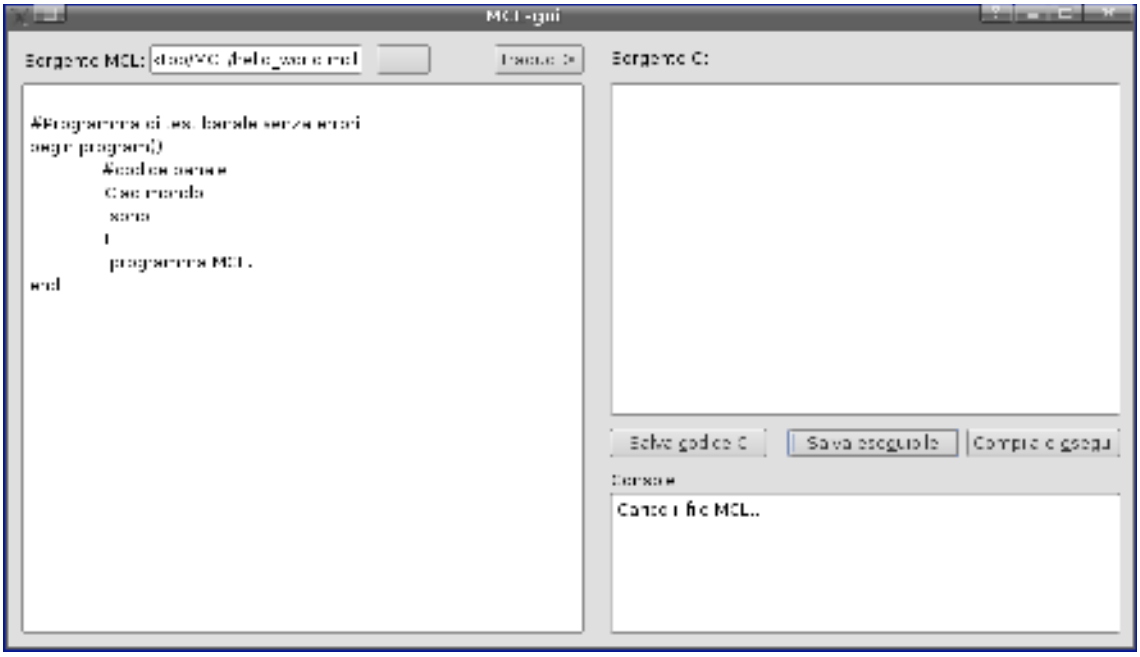


Illustrazione 3: Il codice MCL caricato

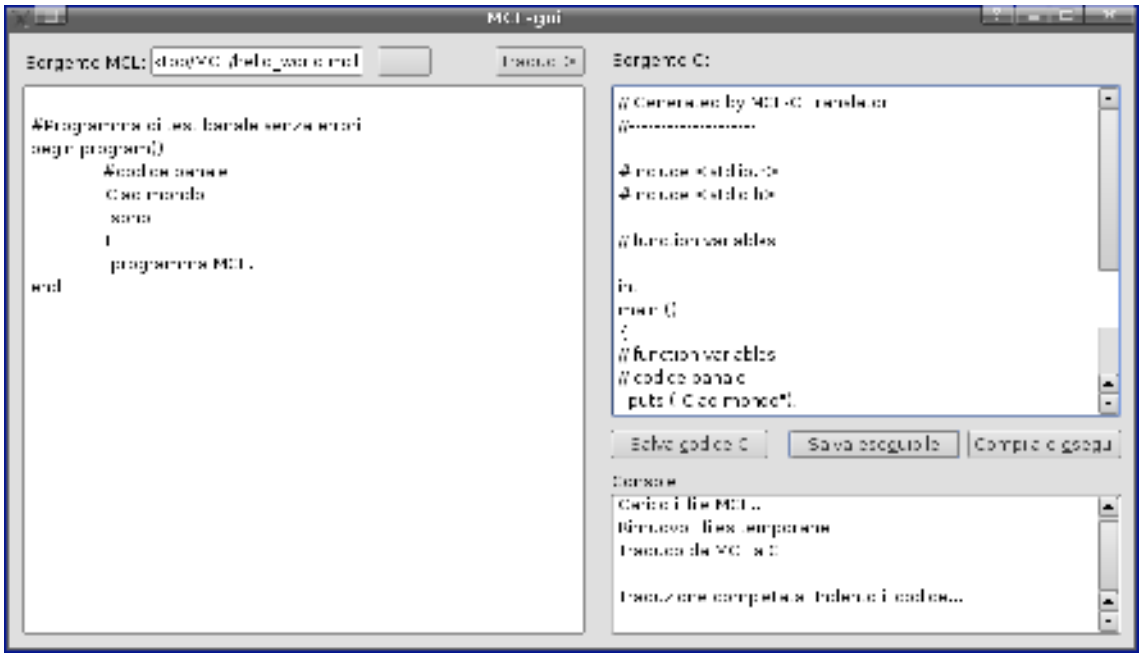


Illustrazione 4: Traduzione da MCL a C

Sul codice C è possibile eseguire tre operazioni: salvare il sorgente C in un file, salvare il file eseguibile o eseguire il programma. Nei primi due casi viene aperta una finestra di dialogo per selezionare la destinazione in cui salvare il file. Cliccando su “Salva eseguibile” o su “Compila e esegui” viene prima avviata la compilazione del codice tramite `gcc`. L'ultimo pulsante avvia infine l'esecuzione del programma all'interno della console.

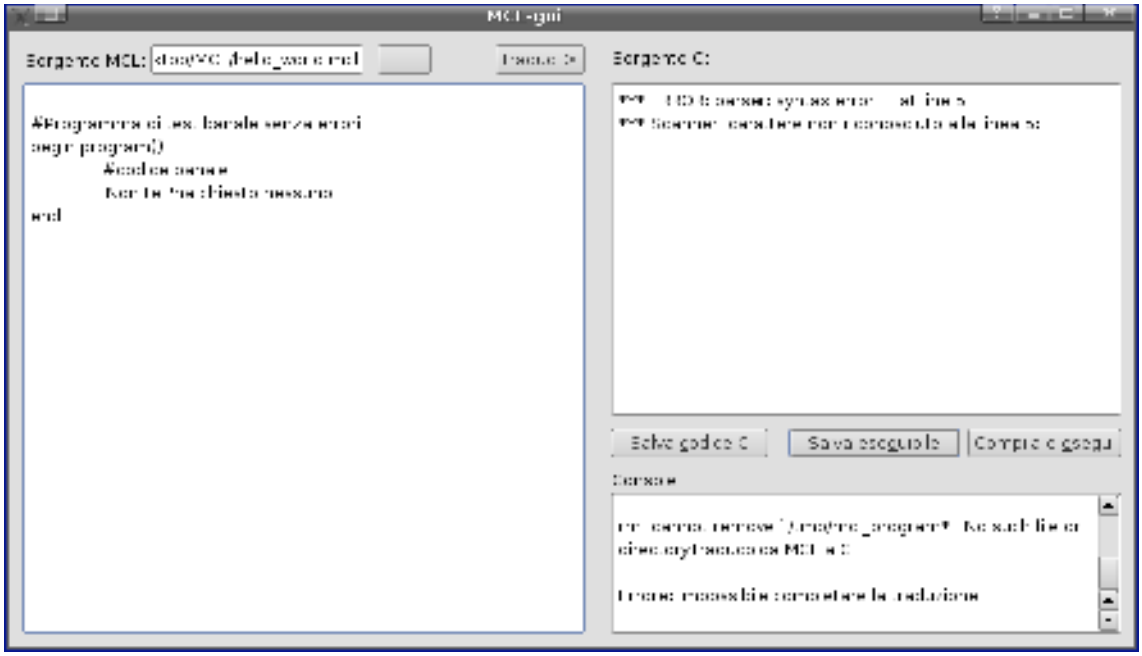


Illustrazione 5: Visualizzazione degli errori di traduzione

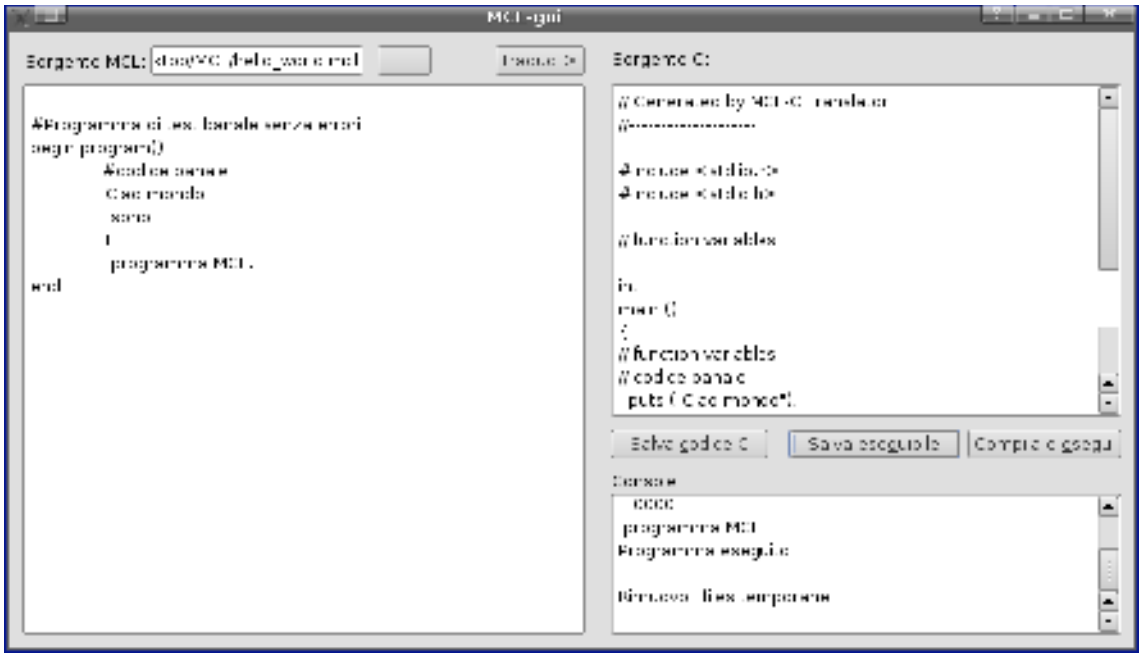


Illustrazione 6: Traduzione in linguaggio C

7.3 Implementazione

Nell'ambiente di sviluppo di Kommander dopo aver definito le caratteristiche della finestra di dialogo è possibile posizionare gli oggetti come pulsanti e aree di testo (detti

widgets). I widget possono essere connessi tramite azioni, in modo che un evento su un widget inneschi un azione su un altro. Nella finestra della finestra di dialogo MCL-gui i click sui pulsanti innescano l'esecuzione di oggetti di tipo script. Questi contengono frammenti di codice scritti secondo una particolare sintassi di Kommander che a loro volta possono eseguire comandi o innescare altre azioni. A titolo di esempio lo script invocato cliccando sul pulsante “Traduci” è come segue:

```
@setGlobal(action,7)
@console.execute()
@exec(sleep 1)
@File.write(/tmp/mcl_program.mcl, @mclCode.text)
@setGlobal(action,2)
@console.execute()
@exec(sleep 1)
@cCode.setText(@File.read(/tmp/mcl_program.c))
```

La prima riga imposta al valore 7 la variabile globale `action` utilizzata dal widget `console` (descritto in seguito). Poi viene avviata l'esecuzione della console e, dopo una pausa, viene scritto nel file temporaneo `/tmp/mcl_program.mcl` il testo contenuto nell'area di testo per il codice MCL. Infine viene avviata nuovamente la console per effettuare la traduzione e copiato nell'area di testo per il codice C il risultato della traduzione.

Quando viene mandato in esecuzione l'oggetto `console`, viene eseguito il seguente script Bash:

```
export TERM=dumb
export ACTION=@global(action)
case $ACTION in
      1)    echo -e "Carico il file MCL...\n";;
      2)    echo -e "Traduco da MCL a C...\n"
            /home/marco/Desktop/MCL/MCL_final/mcl</tmp/mcl_program.mcl
```

```
> /tmp/mcl_program.c 2>&1
            if [ $? = 0 ]
                  then echo -e "Traduzione completata. Indento il codi-
ce...\n"
                        /usr/bin/indent /tmp/mcl_program.c 2>&1
                  else echo -e "Errore: impossibile completare la tra-
duzione\n"
            fi ;;
      3)    echo -e "Salvo il codice C...\n"    ;;
      4)    echo -e "Compilo..."
            gcc /tmp/mcl_program.c -o /tmp/mcl_program 2>&1
            if [ $? = 0 ]
                  then echo -e "Fatto\n"
                  else echo -e "Errore: impossibile compilare\n"
            fi ;;
      5)    echo -e "Salvo il programma...\n"
            mv /tmp/mcl_program @binFile.text 2>&1;;
      6)    echo -e "Eseguo il programma...\n"
            /tmp/mcl_program  2>&1
            if [ $? = 0 ]
                  then echo -e "Programma eseguito\n"
                  else echo -e "Errore: verifica gli input\n"
            fi
            echo -e "Rimuovo i files temporanei...\n"
            rm /tmp/mcl_program* 2>&1;;
      7)    echo -e "Rimuovo i files temporanei...\n"
            rm /tmp/mcl_program* 2>&1;;
esac
```

A seconda del valore assegnato alla variabile globale `action` dell'ambiente Kommander vengono eseguite azioni diverse. Per questo motivo prima di mandare in esecuzione l'oggetto `console`, è necessario assegnare un valore a questa variabile. L'azione 1 e l'azione 3 si limitano banalmente a segnalare attraverso la console l'apertura di un file MCL e il salvataggio del codice C rispettivamente. L'azione 2 chiama il traduttore MCL per effettuare la conversione da MCL a C e scrivere l'output sul file `/tmp/mcl_program.c` . Il valore di uscita di questa operazione viene verificato, in modo da rilevare eventuali errori. Se il programma `mcl` termina correttamente allora si procede all'indentazione del codice tramite il l'utility `indent`, fornita dall'ambiente Linux. In caso contrario viene dato un messaggio di errore. Da notare l'uso dell'operatore `2>&1` inserito per redirigere l'output di errore (stderr) sullo standard output. Si è reso necessario usare questo stratagemma in tutti i comandi dello script perché il widget `console` non ha normalmente la possibilità di visualizzare lo standard error. L'azione 4 è associata alla fase di compilazione. Anche in questo caso si controlla il valore con cui termina il compilatore per individuare eventuali errori e darne correttamente notizia nella console. L'azione 5 serve solamente a copiare il programma (precedentemente compilato attraverso l'azione 4) in una destinazione assegnata attraverso il widget `@binFile.text`. L'azione 6 manda in esecuzione il programma, ne controlla la corretta esecuzione dandone informazione all'utente e, una volta terminato, elimina il files temporanei creati dal traduttore e dal compilatore nella directory `/tmp`. L'ultima azione esegue questa stessa azione in maniera indipendente. Viene chiamata all'avvio degli script di traduzione per partire sempre con un ambiente di lavoro “pulito”.

Una delle limitazioni dell'interfaccia grafica sviluppata sta nel fatto che in Kommander il widget di tipo console non ha la possibilità di gestire l'input. Per questo motivo la funzionalità di esecuzione del codice è da considerarsi puramente a scopi di test su programmi che non prevedono l'interazione dell'utente.

8. Problematiche affrontate di rilievo

Durante lo sviluppo del traduttore per il linguaggio MCL ci siamo trovati a risolvere innumerevoli problemi, sia lato architetturale che implementativo. In questo capito sono messe in evidenza le problematiche di rilievo affrontate e risolte. L'elenco completo dei cambiamenti si trova nel file di CHANGELOG, allegato in capitolo 9.

8.1 La gestione degli errori e il debugging

Durante lo sviluppo è stato necessario implementare un Framework per la cattura degli errori e il debugging, in modo da rendere più agevole la continua evoluzione del codice.

Lo scanner comunica errori nella lettura dei caratteri attraverso l'istruzione

```
.       {
     printf ("*** Scanner: carattere non riconosciuto alla linea %d: %s\n",num_linea, yytext);
                exit (1);
        }
```

segnalando la riga:

*** ERROR: parser: syntax error ... at line 2

Il parser ha un sistema più complesso di gestione degli errori. Ogni funzione quali le int push_funct(char [256], int n);

int push_var(char dato[256], char f[256]);

int push_global_var(char dato[256], u_short in);

giusto per segnalarne alcune ritornano 1 se l'oggetto è stato correttamente inserito in lista, altrimenti 0. Il chiamante si preoccupa di verificare tale valore di ritorno e in caso di errore lo ritrasmette alla funzione padre segnalando un errore. Infine il parser esce con errore 1. Se il processo di traduzione si completa senza errori, il traduttore ritorna 0.

E' lo script di supporto gmcl che controlla l'esito delle singole fasi di compilazione attraverso l'analisi della variabile bash $$.

8.1.1 Comunicazione degli errori e del debug

Gli errori sono comunicati su standard error (stderr) con la funzione fprintf:

```
fprintf (stderr, "ERROR: function %s already declared. Abort!\n", tmp_funct);
```

Il debugging a sua volta viene abilitato dalla variabile di globale MDEBUG e stampato su stderr:

```
#ifdef MDEBUG
fprintf (stderr, "DEBUG: parser PARAM_L: added new _var_ %s (1)\n", $1);
#endif
```

Tale variabile di può essere passata da linea di comando al momento della compilazione del traduttore con -DMDEBUG. Allo stesso modo il debugging interno del parser (bison) si abilita con -DDEBUG.

8.1.2 Esempio di sessione dei debugging di codice con errore semantico.

Nel codice sottostante l'istruzione condiziona è state erroneamente terminata con un punto e virgola:

```
begin program(a)
      if (a>0)
         'codice if'
      fi;
end
```

E' necessario abilitare il debugging del Bison e quello del traduttore mcl in fase di compilazione, con -DDEBUG e -DMDEBUG:

```
$ gcc -DDEBUG -DMDEBUG -g -O2 -Wall lex.yy.c mcl.tab.c -lfl -o mcl
```

Ed ecco la sessione di debugging del codice:

```
DEBUG ENABLED with -DMDEBUG
Starting parse
Entering state 0
Reading a token: Next token is token MBEGIN ()
Reducing stack by rule 4 (line 108), -> FINE_L
Stack now 0
Entering state 3
Next token is token MBEGIN ()
Reducing stack by rule 2 (line 100), -> CL
Stack now 0 3
Entering state 7
Next token is token MBEGIN ()
Reducing stack by rule 6 (line 113), -> VL
DEBUG: parser push_funct(): adding function 255 with #param = 0
DEBUG: parser check_funct(): checking _function_ 255 in the list
DEBUG: parser check_funct(): [KO] function not found!
DEBUG: parser push_funct(): [OK] added function 255
Stack now 0 3 7
Entering state 10
Reducing stack by rule 8 (line 135), -> PL
Stack now 0 3 7 10
Entering state 13
[cut]
Next token is token PUNTOVIRGOLA ()
*** ERROR: parser: syntax error ... at line 4
```

```
Error: popping nterm FINE_L ()

Stack now 0 3 7 10 13 15 19 25 33 54 30 46 72

Error: popping nterm FINE_L ()

Stack now 0 3 7 10 13 15 19 25 33 54 30 46

Error: popping nterm INSTR ()

Stack now 0 3 7 10 13 15 19 25 33 54 30

Error: popping nterm FINE_L ()

[cut]

Cleanup: discarding lookahead token PUNTOVIRGOLA ()
```

Nota bene: Le righe contenenti del parole di codice DEBUG e ERROR sono scritte dal framework del traduttore mcl (MDEBUG).

8.2 Gestione delle informazioni in memoria

La traduzione di un linguaggio in un altro necessita della scrittura di opportune azioni semantiche per la rappresentazione in memoria del codice sorgente in un opportuno schema logico. Durante la sintesi delle produzioni il parser inserisce in memoria il codice MCL e in una seconda fase ne riestrae l'intero contenuto traducendolo nella nuova forma di destinazione.

Le differenze tra i due linguaggi impediscono una traduzione on-the-fly. Basti pensare che in MCL è possibile dichiarare le variabili al momento del loro utilizzo, mentre in C questo deve essere fatto all'inizio della funzione. In questo esempio si vede come sia necessario andare prima (nella fase di analisi semantica) a recuperare tutte le variabili utilizzate, inserirle in memoria e solo in un secondo momento, durante la fase di traduzione, scriverle in blocco all'inizio di ciascuna funzione.

Inoltre un altro problema è la caratteristica intrinseca del bison di sintetizzare le produzioni dal basso verso l'alto. In un contesto in cui bisogna rappresentare per ciascuna funzione le variabili che riceve in input, vuol dire che il parser sintetizza prima le varia-

bili e via via la funzione. Le azioni semantiche devono essere quindi in grado di allocare spazio alle variabili per una funzione non ancora sintetizzata.

La sfida più grande è stata quindi quella di costruire e gestire l'albero semantico in memoria, implementando le funzioni di push_ per l'inserimento delle informazioni e quelle di print_ per la loro estrazione. Durante le operazioni di push_ sono stati implementati controlli sull'esistenza di copie uguali di informazioni. Le funzioni di check_ si preoccupano di tale analisi ritornando valore uno in condizione di errore. In questo il traduttore MCL scopre dichiarazioni doppie di variabili e funzioni, già prima di tradurre in codice in C. In una prima istanza noi sviluppatori avevamo deciso di lasciare questo compito al compilatore C, ma poi è stato preferito inserire questo tipo di controlli direttamente nella fase di pre-processing. In un futuro si vorrà introdurre il supporto all'overloading, permettendo la dichiarazione di funzioni con lo stesso nome ma numero dei parametri differenti. Sarà necessario modificare la struttura nodo delle funzioni per aggiungere un campo di identificazione differente dal quello del nome usato in questa versione.

E' stata fatta molta attenzione all'ordine con cui vengono sintetizzate e conseguentemente inserite le informazioni. In alcuni casi è stato opportuno utilizzare un inserimento in testa, invertendo l'ordine delle informazioni. Sempre per la natura down-top del Bison le variabili di input alle funzioni sono sintetizzate in ordine inverso e quindi inserendole in testa otteniamo un ordinamento corretto.

9. Il diary file (Changelog)

Quello riportato ora è un changelog che riporta le varie azioni intraprese per la realizzazione del compilatore/traduttore, con le varie modifiche effettuate nel corso dello sviluppo del progetto.

6 marzo – 13 marzo

Analisi dei vari strumenti da utilizzare per la realizzazione del progetto, con particolare riferimento ai pro e ai contro.

Scelta del progetto da realizzare e primo raffronto tra i possibili strumenti da utilizzare e le specifiche del nostro progetto, in particolare riguardo al tipo di grammatica del progetto (se LR(0) o LALR(1)...)

20 marzo -27 marzo

Stesura della grammatica del linguaggio scelto e di un esempio funzionale, per capirne meglio le caratteristiche

3 aprile

Prima stesura di uno scanner con lo strumento *Flex* (l'analizzatore lessicale).

Corretto i vari errori di sintassi riscontrati nello scanner

Realizzato “scanner di test”. E' una versione dello scanner che invece di passare i vari token al parser, stampa a video i token che incontra nella scansione di un programma scritto in MCL, per vedere se riconosce correttamente ciò che gli viene passato

10 aprile

Revisione scanner

Prima stesura del parser: scritta l'intera grammatica BNF di MCL in un linguaggio comprensibile per lo strumento Bison (il nostro parser generator)

17 aprile

Continuato con la stesura del parser

Revisione finale del parser (per quanto riguarda la grammatica)

8 maggio

Corretto conflitti flex e bison Vs cpp: i token che lo scanner restituiva venivano passati al parser che, essendo basato sul C, interpretava alcuni token come key-word secifiche del C (come ad esempio i costrutti *if, else, while*...)

Eliminata gestione carattere FINELINEA (`'\n'`) per problemi con la terminazione della scansione del file

Aggiunto riconoscimento del TAB nello scanner: senza di questo lo scanner restituisce errore perchè non riconosce il carattere

Sistemata gestione commenti nel codice: lo scanner restituiva errore se trovava dei caratteri illegali in un commento

Creato e provato due programmi di test

11 maggio

Aggiunta possibilità di compilare con debug

Corretto bug nel caso in cui si scriva un'istruzione contenente un solo identificatore (senza ;)

15 maggio

Corretto errori parser: condizioni if e while definite in modo errato nella grammatica

Corretto programma di test2 con blocchi if e while

Aggiunto contatore linea per errori dello scanner (on-error, viene restituita anche la linea in cui si trova l'errore)

Aggiunto contatore linea per errori del parser (come sopra)

Corretto errore doppi apici all'interno di stringhe e commenti

Iniziata "trasformazione in c di mcl": scelta della struttura dati ad albero

18 maggio

Effettuati test per verificare le correzioni precedenti

Tolto obbligo del punto virgola dopo "exit"

Creazione albero semantico per conversione da mcl a C

* Creazione main: [OK]
* Conversione commenti: [OK]
* Conversione var globali: [OK]
* Conversione parametri main: [OK]

Messo basi per conversione BODY e lista delle funzioni

19 maggio

Continuazione albero semantico di conversione

* Conversione funzioni (solo dichiarazione) [OK]

Test su quanto fatto finora

22 maggio

Continuazione albero semantico di conversione

* Conversione frasi [OK]
* Conversione chiamate a funzioni [OK]

* Conversione blocchi if [OK]

* Conversione blocchi while [OK]

* Conversione condizioni di confronto [OK]

* Iniziata conversione espressioni [OK]

Aggiornata relazione con ultime modifiche allo scanner e al parser

23 maggio

Continuazione albero semantico di conversione

* Conversione espressioni [OK]

* Gestione punto virgola(stampa/non stampa il risultato) [OK]

29 maggio

Aggiunto riconoscimento '_' nei nomi di variabili e di funzione

Aggiunta possibilità di scrivere espressioni con segno (+ -)

Iniziato controllo dichiarazione variabili e funzioni

Riflessione sull'importanza della gestione del FINELINEA

Iniziato sviluppo interfaccia grafica

5 giugno

Aggiunta gestione FINELINEA: alla fine è stata aggiunta la gestione del FINELINEA perché due espressioni consecutive (di cui la prima senza punto virgola) venivano gestite come se fossero una sola espressione

Prima realizzazione della lista di liste per la gestione delle variabili e delle funzioni

Iniziato a stilare relazione definitiva

6 giugno

Riscritta la push_funct e check_funct

Sistemato l'input da command line (argv, argc)

Cambiati i %d nei printf con %.2f

Normalizzati i formati di debug

Corretta la check_var

Nuove routine di aggiunta variabili e funzioni

Riscritta la push_var che trova la funzione

int print_function_vars(char f[256]);

Sistemata la copia delle liste da tmp_var_list a var_list

(il malloc + memcpy funziona solo con spazi di memoria continui come vettore e non liste dinamiche)

Aggiunto nei nodi della variabili il flag 'in' così è possibile riconoscere i tipi di variabili (variabili locali o globali/parametro)

7 giugno

Correzione doppioni variabili:

* variabile di ritorno delle funzioni duplicata [OK]

* doppioni variabili globali – locali [OK]

Correzione "Segmentation Fault" nelle condizioni if & while

Corretto parentesi graffe nel main

Aggiunto controllo sul numero dei parametri in una chiamata di funzione (concorde con definizione)

Aggiunto controllo sulla dichiarazione delle funzioni (verifica esistenza funzioni in fase di inserimento e quando vengono richiamate)

Appendice A Riferimenti

- GNU Flex, http://www.gnu.org/software/flex/
- GNU Bison, http://www.gnu.org/software/bison/
- GNU C Compiler, http://gcc.gnu.org/
- GNU/Linux, http://www.kernel.org/

Appendice B Sorgenti

Di seguito sono riportati i seguenti sorgenti:

- Scanner Flex, mcl.l
- Parser Bison, mcl.y
- Script di compilazione del traduttore, build-mcl
- The GNU MCL Compiler, gmcl