

1.1.1 Preprocessori e Linguaggi Macro

I Preprocessori sono strumenti di elaborazione automatica di testi; nel campo dei linguaggi di programmazione sono stati utilizzati o per trasformare sorgenti da un linguaggio ad un altro o per aggiungere funzionalità a linguaggi esistenti.

1.1.1.1 gema

gema (*general purpose macro processor*) è un text processor di David N. Gray creato nel 1995 e successivamente rivisto nel 2003.

gema effettua delle sostituzioni di stringhe secondo quanto indicato da una serie di regole (*template*) aventi la forma:

```
sostituendo=sostituente[sostituendo=;sostituente...]
```

dove *sostituendo* può essere un'insieme di caratteri, o un *pattern* che va dai semplici caratteri jolly * e ?, fino alle espressioni regolari, passando per un buon numero di pattern particolari la cui forma è *<In>* in cui il valore numerico opzionale *n* indica la quantità totale (lettera maiuscola) o massima di caratteri (lettera minuscola), individuata dal tipo *I*; l'eventuale segno - davanti alla lettera, nega la condizione.

<A>	Lettere e digits
<D>	Digits
<F>	Filename
<O> <X>	Ottale e esadecimale
<L>	lettere
<J> <K>	Lettere minuscole e maiuscole
<N>	Numeri segnati con separatore decimale
<I>	Identificatore: <A> più underscore
<U>	Tutto (tranne il fine file).

Il carattere \ è il carattere di *escape* per individuare gli usuali caratteri non stampabili e altri punti del testo fra i quali \A e \B per l'inizio del file di testo, \E e \Z per la fine, \s uno spazio.

Il risultato del match relativo ai pattern è conservato nelle variabili \$1, \$2, ..., mentre la riga di testo è individuata dalla variabile \$0; nell'esempio sottostante gema costruisce un'istruzione (COBOL):

```
comando  gel -p "<N4> <n>=ADD $1 TO $2 GIVING SUM"
input    1.22 -2344.5
output   ADD 1.22 TO -2344.5 GIVING SUM
```

Nella parte sostituente, oltre alla presenza di caratteri e variabili estratte, è possibile inserire il risultato ottenibile da un vasto numero di funzioni disponibili. Le funzioni numeriche sono limitate ai numeri interi relativi, qui di seguito la regola per eseguire una divisione:

```
gel -p "<n> <n>\n=$1 \\ $2 \= @div{$1; $2}\n"
-99 -9
-99 \ -9 = 11
-99.9 -11
-99.9 \ -11 = Non-numeric operand: "-99.9"
```

Si noti, inoltre, che per catturare il segno occorre utilizzare il pattern <n>, che di per se accetta anche il separatore decimale, ma con il separatore decimale la variabile catturata non è considerata un numero.

Anche le funzioni su stringhe di caratteri sono numerose, con particolare riguardo a quelle di formattazione, ad esempio per aggiungere una numerazione allineata ad un file di testo:

```
\L<u>\n=@right{5;@sub{@line;1}}\s$0\n1;
```

Esistono funzioni per operare su file, per memorizzare variabili, per gestire le regole e due, @cmpn e @cmps, per fornire alternative (<, = e >), in base a confronto fra numeri o stringhe.

E' possibile dare un nome alle regole con la sintassi: *nome:regole*, e queste si richiamano e si comportano come funzioni, rendendo possibile l'applicazione selettiva di regole, ad esempio per elaborare ulteriormente una stringa estratta; inoltre con la sintassi *nome₁:nome₂nome₁* eredita tutte le regole di *nome₂*.

L'esempio che segue calcola il Massimo Comun Divisore di due numeri:

```
! separo i due numeri iniziali
<d> <d>\n=@gcd{$1 $2}
gcd:<d> <d>=@cmpn{$1;$2;@gcd{@sub{$2;$1} $1};$1\n;@gcd{@sub{$1;$2} $2}}
```

Come si può vedere gema supporta la ricorsività.

¹ La funzione @line, nella versione Windows di gema, restituisce il valore della riga più 1.

L'inclusione in gema di Lua versione 5 offre ulteriori capacità di calcolo.

Nell'esempio, avendo in input delle righe come le seguenti:

```
0101#Maria Ss. Madre di Dio#
0102#Auguri a Basilio e Gregorio!#
0103#Auguri a Daniele (di Padova)!#
0202#Present. del Signore#
0227#Auguri a Onorina!#
0228#Auguri a Romano e Osvaldo!#
```

le regole sottostanti :

```
!GUA 1.0 Lua+GEMA Copyright (C) 2004 David N. Gray
!Lua 5.0 Copyright (C) 1994-2003 Tecgraf, PUC-Rio
! gema 1.4.1RC Mar 31, 2005
\B=-- @repeat{20;*} SQL Commands for inserting data @repeat{20;*}\n
\L<U4>\#<u>=INSERT INTO Santi (Giorno,Santo)
VALUES(\'$1\','@replace{@lua{return Sant("$2")}}\')\;
!clean:Auguri a =
! (, ), #, ! and ' replacements
replace:\s\(\s\(\s\(\s\)=;\=''\';\#=\!:=;
\Z=\n-- Worked\s@line records
![
function Sant(inp)
    if string.sub(inp, 1, 8) == "Auguri a" then
        if string.find("AEIOU",string.sub(inp,10,10),1,true) == nil
then
            return "San"..string.sub(inp, 9)
            else
            return "Sant'"..string.sub(inp, 10)
            end
        else
        return inp
        end
    end
end
!]
```

Producono una serie di comandi SQL:

```
I:\GEMA>gel -f rules.txt santosprova.txt
-- ***** SQL Commands for inserting data *****
INSERT INTO Santi (Giorno,Santo) VALUES('0101','Maria Ss. Madre di Dio');
INSERT INTO Santi (Giorno,Santo) VALUES('0102','San Basilio e Gregorio');
INSERT INTO Santi (Giorno,Santo) VALUES('0103','San Daniele di Padova');
INSERT INTO Santi (Giorno,Santo) VALUES('0202','Present. del Signore');
INSERT INTO Santi (Giorno,Santo) VALUES('0227','Sant''Onorina');
INSERT INTO Santi (Giorno,Santo) VALUES('0228','San Romano e Osvaldo');

-- Worked 7 records
```

1.1.1.2 M4

Macro processore 1977 -

È un non molto sofisticato preprocessore di derivazione UNIX dovuto a Brian Kernighan e Dennis Ritchie.

E' tuttora usato anche in Window, ed in evoluzione, la release 1.4.14 è del 2010 (Eric Blake).

Il suo uso va dal preprocessare sorgenti di compilatori, è stato usato per Ratfor, C and Cobol, alla generazione di pagine HTML.

M4 analizza il testo in ingresso e lo separa in *tokens*. Un token può essere un nome di macro, una stringa tra virgolette, o un qualsiasi carattere, che non appartenga ad un nome o ad una stringa; se il token non corrisponde ad una macro, interna o predefinita, esso è mandato in uscita, altrimenti questa è eseguita ed il suo l'output farà parte immediatamente dei dati di ingresso.

I commenti sono un effetto secondario del comportamento di M4: la macro `dnl` ha l'effetto principale di ignorare il resto della riga di testo per evitare l'invio in uscita di righe vuote; il carattere `#` ha l'effetto di inviare in uscita di quanto è compreso fra se stesso ed il fine riga, senza sottoporlo a trasformazione.

In M4 è quasi impossibile gestire variabili per memorizzare informazioni durante la generazione dell'output.

Qui di seguito l'insieme di macro per generare programmi di una macchina virtuale che ha la sola istruzione di sottrazione con salto per risultato negativo; la struttura di tale macchina è:

Nelle prime celle sono contenuti:

- **PC** il Program Counter
- **R0, R1** celle di lavoro o registri
- **Input** buffer dei dati di input
- **Output** buffer dei dati di output
- **Uno** la costante 1
- **A, B,...** celle di lavoro

Seguono le istruzioni della macchina ognuna delle quali occupa tre celle consecutive C_1, C_2, C_3 , l'esecuzione del programma è:

$$C_1 = C_1 - C_2$$

$PC = PC - C_3$ se C_1 è diverso da 0, altrimenti $PC = PC + 3$.

```
dnl GNU m4 1.4 file unaistr.m4
dnl mettiamo dei delimitatori seri: []
changequote([, ])
dnl macro per commentare le istruzioni
define([commento],[* [$1]($2) * $3])dnl
dnl scan serve per generare l'immagine della memoria
define([scan], [
  ifelse($#, 0,, $#, 1, genfield($1), [genfield($1)
scan(shift($@))])dnl
])dnl
dnl genera la memoria di lavoro
define([genfield],[
  define([valore],[ifelse($1,PC,TotMem,ifelse(index($1,[=]),
-1,0,substr($1,incr(index($1,[=])))))]dnl
dnl contenuto della memoria iniziale
format([:%s %4d],ifelse(index($1,[=]),
-1,$1,substr($1,0,index($1,[=])),valore)
])dnl
define([MEM],[commento([$0],[@$],[contenuto della memoria])
define([TotMem],$#)dnl
scan($@)dnl
])dnl
```

```

define([ALT],[commento([$0],[@$],[fine lavoro])
PC PC -3
])dnl
define([GET],[commento([$0],[@$],[lettura dati: Input -> $1])
R0 R0 -3
R0 Input -18
R1 R1 -3
R1 R0 -6
R1 Uno 12
R0 R0 -3
R0 R1 -3
$1 $1 -3
$1 R0 -3
Input Input -3
])dnl
define([PUT],[commento([$0],[@$],[scrittura dati: $1 -> Output])
R0 R0 -3
R0 Output 3
R1 R1 -3
R1 R0 9
MOVE(Output,$1)
])dnl
define([MOVE],[commento([$0],[@$],[spostamento dati: $2 -> $1])
R0 R0 -3
R0 $2 -3
$1 $1 -3
$1 R0 -3
])dnl
define([DIV],[commento([$0],[@$],[divisione: $1/$2 -> $1])
R0 R0 -3
$1 $2 -6
R0 Uno 3
$1 $1 -3
$1 R0 -3
])dnl

```

Il "programma" qui di fianco utilizza le macro definite nel precedente listato (che è stato incluso mediante l'istruzione `include(unaistr.m4)`). Il programma genera l'immagine della memoria della macchina contenente le istruzioni per dividere due numeri interi positivi.

```

include(unaistr.m4)
MEM(PC,R0,R1,Input,Output,Uno=1,A,B)
GET(A)
GET(B)
DIV(A,B)
PUT(A)
ALT

```

L'espansione in "linguaggio macchina" è listata qui di seguito (a parte l'eliminazione manuale delle righe superflue):

```

* MEM(PC,R0,R1,Input,Output,Uno=1,A,B) * contenuto della memoria
:PC      8
:R0      0
:R1      0
:Input   0
:Output  0
:Uno     1
:A       0
:B       0
* GET(A) * lettura dati: Input -> A
R0 R0 -3
R0 Input -18
R1 R1 -3
R1 R0 -6
R1 Uno 12

```

```

R0 R0 -3
R0 R1 -3
A A -3
A R0 -3
Input Input -3
* GET(B) * lettura dati: Input -> B
R0 R0 -3
R0 Input -18
R1 R1 -3
R1 R0 -6
R1 Uno 12
R0 R0 -3
R0 R1 -3
B B -3
B R0 -3
Input Input -3
* DIV(A,B) * divisione: A/B -> A
R0 R0 -3
A B -6
R0 Uno 3
A A -3
A R0 -3
* PUT(A) * scrittura dati: A -> Output
R0 R0 -3
R0 Output 3
R1 R1 -3
R1 R0 9
* MOVE(Output,A) * spostamento dati: A -> Output
R0 R0 -3
R0 A -3
Output Output -3
Output R0 -3
* ALT() * fine lavoro
PC PC -3

```

L'interprete del linguaggio, con un esempio di calcolo, è scritto in Ruby, con l'utilizzo del suo facile meccanismo di implementazione della concorrenzialità.

1.1.1.3 ML/I

Macro processore 1966 -

ML / I (*Macro Language One*) è stato sviluppato come parte del dottorato di ricerca di Peter Brown nel 1966. ML1 è nato su Digital PDP-7, e sul sistema Titan a Cambridge. Nel tempo il programma è stato portato su una trentina di implementazioni fra cui su PDP-11, OLIVETTI L1 e ICL 4130, partendo dal sorgente scritto in un linguaggio astratto chiamato L e un assembler LOWL (Low level L). Fra le evoluzioni più significative che ha avuto nel tempo, l'introduzione dello pseudocarattere SL (*startline*) per rendere più facile l'elaborazione di testi basati su insiemi di linee, e l'accettazione di variabili carattere.

ML/I opera su un insieme di caratteri (*stream*), senza un particolare formato dell'input o delle macro di trasformazione; le macro sono contenute nel testo di input (tuttavia la possibilità di indicare fino a 5 file di input permette di separare le macro dal testo).

ML/I agisce su un testo che è formato da atomi: parole, numeri o caratteri speciali, NL (*new line*) è l'indicazione di fine riga. Il meccanismo di trasformazione si basa sul riconoscimento di delimitatori, di inizio, eventuali intermedi e fine, ad esempio FOR = TO OPT STEP è lo schema di riconoscimento del FOR Basic. Il delimitatore iniziale è il nome della macro; questa caratteristica è anche una limitazione in quanto solo le parti comprese fra i delimitatori possono essere riconosciute per tipo (*Class*), peraltro limitato a numeri relativi, parole ed identificatori.

Ci sono una trentina di macro di sistema, riconoscibili perché iniziano con MC, fra le più significative:

- MCSKIP usata per eliminare del testo compreso fra due delimitatori, ad esempio con MCSKIP < > si eliminano i tag HTML; un uso tuttavia più significativo è quello di inibire l'esecuzione di una macro all'interno di un testo o di una espansione di macro. Infine, poiché le macro non sono modificabili, MCSKIP ne permette l'eliminazione, ad esempio dopo MCSKIP D , < i TAG HTML non saranno più eliminati dal testo.
- MCINS permette di inserire il valore delle variabili nel testo in uscita; la macro ha come operandi i due delimitatori con i quali saranno racchiuse le variabili, esempio: MCINS %..
- MCDEF definisce uno schema di riconoscimento di testo: MCDEF *schema* AS *sostituzione*.

```
->) MCSKIP < >
->) MCSKIP T, [ ]
->) <DIV>[<B>Bold<B/>]</DIV>
<-) <B>Bold<B/>
```

Un esempio di macro MCDEF per trasformare l'istruzione COMPUTE del COBOL nell'equivalente espressione C è:

```
->) MCDEF COMPUTE = NL AS [%A1. = %A2.; %D2.]
->) COMPUTE SCONTO = (IMPORTO + BOLLO)* 0.88
<-) SCONTO = (IMPORTO + BOLLO)* 0.88;
```

COMPUTE diventa il nome della macro, gli altri delimitatori sono = e NL; A1 e A2 sono le variabili che contengono il testo compreso fra i delimitatori e D2 contiene il carattere di NL.

- MCSUB e MCLENG forniscono rispettivamente una sottostringa e la lunghezza di una stringa data.
- MCSET per assegnare una valore ad una variabile.
- MCGO permette di variare la sequenza di esecuzione delle istruzioni, accetta come parametri opzionali IF e UNLESS con le condizioni BC (*Belongs to Class*), EN (*Equals Numerically*), GE e GR.

```
MCDEF THIS NL
AS [MCSET C10 = Numeric
MCGO L1 IF %A1. BC N
MCSET C10 = Word
MCGO L1 IF %A1. BC L
MCSET C10 = Identifier
MCGO L1 IF %A1. BC I
MCSET C10 = Unknown
%L1. %A1. IS %C10. %D1.]
-> THIS Condor
-> Condor IS Word
-> THIS -31415
-> -31415 IS Numeric
-> THIS P38MAGNUM
-> P38MAGNUM IS Identifier
-> THIS $**&/
-> $**&/ IS Unknown
```

ML/I prevede un insieme di variabili: da quelle generate durante il riconoscimento di uno schema, a quelle

MCPVAR <i>n</i>	MCPVAR 100 MCSET P100 = 30
MCCVAR <i>n, lunghezza</i>	MCCVAR 15, 20 MCSET C15 = CONDOR INFORMATIQUE

di sistema contenenti dei parametri del programma, a quelle dell'utilizzatore da dichiarare tramite le macro MCCVAR e MCPVAR e valorizzare con la macro MCSET:

- A0, A1, ... parte di testo contenuta fra i delimitatori,

- D1, D2,... delimitatori,
- P1, P2, ... C1, C2, ... variabili permanenti a disposizione, le variabili C_n memorizzano stringhe di caratteri.
- S1, S2, ... S24 per il controllo di ML/I, per esempio S16 e S17 sono usate per trascodificare caratteri, S2 contiene il numero di linea in elaborazione.
- T1, T2 e T3, variabili temporanee generate durante l'esecuzione della macro, in particolare T1 contiene il numero di argomenti della macro corrente.
- L1, L2,...etichette, usate nella macro MCGO.

```
MCDEF = OPT ** OR ^ ALL NL
AS [MCSET C10 = %A2.
MCSET C12 = %DT1.
MCGO L2 IF %D1. = ^
MCSET C10 = %A3.
%L2.%C11. power(%A1.,%C10.)
%C12.]
```

Qui a lato una macro per trasformare la notazione infissa dell'operatore elevazione a potenza (** o ^) in una funzione di due variabili. Da notare OPT unito ad ALL per indicare la presenza di uno dei due operatori. T1 contiene il numero di variabili presenti e di conseguenza la variabile C12 con %DT1. cattura il NL, ciò è dovuto al

fatto che ** per ML/I sono due delimitatori.

Il risultato dell'esecuzione della macro è:

```
-> A = 3**4
<- A = power(3,4)
-> C = A^C
<- C = power(A,C)
```

1.1.1.4 Altri

GPM (the *General Purpose Macroprocessor*) Strachey 1965 University of Cambridge, UK. Usato per tradurre in linguaggio macchina un sottoinsieme di CPL. Una variante di GPM, a nome **BGPM**, è inclusa nella distribuzione di BCPL. GPM è stato un antenato di M4.

M3 Dennis Ritchie macro processore per il minicomputer AP-3.

TMG (*TransMoGrifiers*) Mc Clure 1965. Utilizzato per PL/I di Multics.

STAGE2, William M. Waite per il *Mobile Programming System*. E' un processore di macro in cui ogni riga può essere modificata secondo dei modelli specificati.

XPOP Mark Halpern in IBM anni 60; un tentativo di un macro linguaggio.

PL360 è un compilatore assembler per sistemi IBM 360 e 370 con macro per renderlo strutturato.

GPL (*Genken Programming Language*) è una variante di PL360.

TILT (*Texas Instrument Language Translator*) Anni 70-80 Implementazione di STAGE2 di Roger Hall.

groff (GNU Troff) è un programma di composizione tipografica che legge il testo normale che contiene comandi di formattazione. Groff 1.20.1 supporta HTML.