Grammar Reader

Dispensa di Laboratorio di Linguaggi di Programmazione

A.A. 2005-2006

Corrado Mencar, Pasquale Lops

Sommario

Lo scopo di questo caso di studio consiste nello sviluppo di un programma, in linguaggio C, per la lettura di una generica grammatica da un file di testo e per la sua memorizzazione in una opportuna struttura di dati utile a successive elaborazioni.

Introduzione

Si procede dapprima all'analisi del problema, che prevede un'opportuna formalizzazione delle grammatiche che si dovranno acquisire. In questa fase sono esplicitate le assunzioni semplificative per rendere più semplice lo sviluppo del programma di lettura.

Successivamente, si procede alla definizione della soluzione implementativa. In questa fase si definiscono le strutture dati che conterranno le grammatiche acquisite da file. Sulla base delle strutture dati definite, si procede alla definizione di un insieme di procedure che consentono la loro gestione. Infine, viene individuata la procedura di acquisizione della grammatica, che ha il compito di popolare la struttura dati adibita alla memorizzazione della grammatica memorizzata in un file di testo.

Analisi del problema

La formalizzazione di una grammatica prevede la definizione di una quadrupla:

$$G = (X, V, S, P)$$

dove:

- X è l'alfabeto dei simboli *terminali*;
- V è l'alfabeto dei simboli *non terminali*;
- $S \in V$ è il *simbolo di partenza*
- P è l'insieme delle *produzioni*;

con l'assunzione aggiuntiva che $X \cap V = \varnothing$. L'insieme P di produzione è definito da *regole di produzione* di tipo:

$$\alpha \to \beta$$

dove:

- α è una parola in $(X \cup V)^+$. Questo significa che α non può essere la parola vuota;
- β è una parola in $(X \cup V)^*$. β può dunque essere eventualmente vuoto;
- $\to$ è un separatore $\to \notin X \cup V$

Assunzioni semplificative

La rappresentazione di una grammatica in un calcolatore può essere piuttosto complessa se non si prevedono opportune assunzioni. Le assunzioni semplificative che si adottano in questo caso di studio sono:

1. L'alfabeto dei simboli terminali è un sottoinsieme delle lettere minuscole dell'alfabeto: $X \subseteq \{a, b, ..., z\}$;
2. L'alfabeto dei simboli non terminali è un sottoinsieme delle lettere maiuscole dell'alfabeto: $V \subseteq \{A, B, ..., Z\}$
3. Il simbolo separatore viene rappresentato con il simbolo '>';
4. L'insieme delle produzioni viene rappresentato da un elenco. Le regole di produzioni sono separate dal "ritorno a capo";
5. L'alfabeto dei simboli terminali viene determinato automaticamente dall'insieme delle produzioni: esso è definito da tutte e sole le lettere minuscole che compaiono nelle regole di produzione;
6. Il simbolo di partenza S corrisponde alla lettera maiuscola 'S';
7. Analogamente, l'alfabeto dei simboli non terminali è definito dall'insieme di tutte le lettere maiuscole che occorrono nelle regole di produzione, eventualmente aumentato dal simbolo di partenza S.

Queste assunzioni esemplificative consentono di rappresentare una grammatica in un file di testo, codificato in ASCII, semplicemente come un insieme di linee di testo, in cui ciascuna linea definisce una regola di produzione.

Esempio di rappresentazione di una grammatica

S>aB B>bB B> B>aA A> A>aA A>bB	$X = \{a, b\}$ $V = \{A, B, S\}$ $P = \begin{cases} S \to aB \\ B \to bB \\ B \to \lambda \\ B \to aA \\ A \to \lambda \\ A \to aA \\ A \to bB \end{cases}$

Progettazione della soluzione

La progettazione della soluzione prevede dapprima una fase di definizione della **struttura dati** più appropriata per la memorizzazione di una grammatica caricata da un file testuale. Successivamente si procede alla definizione di un insieme di procedure legate alla gestione di tale struttura (p.e. la visualizzazione su schermo delle informazioni in essa contenute). Infine, si delinea la procedura principale per il caricamento di una grammatica da un file testuale.

L'approccio alla progettazione adottato è di tipo *bottom-up*. Tale approccio prevede la progettazione delle strutture e dei moduli a partire dagli elementi di maggiore dettaglio per arrivare alla definizione di strutture e procedure più generali.

Simboli

Struttura dati

Durante la fase di analisi è stato assunto che sia i simboli terminali, sia quelli non terminali, siano rappresentati da caratteri (rispettivamente minuscoli e maiuscoli). Si può pertanto ritenere opportuno definire il seguente tipo di dati per la rappresentazione dei simboli:

```
01 typedef char Symbol;
```

Il tipo `Symbol` è dunque definito dal tipo predefinito **`char`**. La definizione del tipo `Symbol` è utile perché l'assunzione di rappresentare un simbolo con un carattere ASCII potrebbe essere rimossa in futuro, cosicché una versione successiva del programma potrebbe gestire simboli rappresentati da più caratteri (per esempio, si potrebbe rappresentare il simbolo x_{12} con la sequenza di caratteri ASCII `x_12`). Quando tale estensione dovrà essere implementata, sarà sufficiente cambiare opportunamente la definizione del tipo `Symbol` (insieme a tutte le procedure di gestione di questo tipo dati, che accedono pertanto alla sua definizione), lasciando invariate le implementazioni di tutte le procedure che *usano* il tipo `Symbol` senza accedere alla sua definizione.

Procedure e funzioni

Lettura di un simbolo da file

Innanzitutto si definisce una funzione per la lettura di un simbolo da file. Tale procedura riceverà in input un puntatore a file e restituirà un valore di tipo `Symbol`. La dichiarazione (*signature*) della funzione è dunque:

```
02 Symbol read_sym(FILE* file)
```

Questa funzione carica un simbolo dal file specificato in input. Essa dunque dovrà necessariamente accedere alla definizione del tipo `Symbol`. La funzione è definita semplicemente come segue:

```
{
        Symbol s;

        do
          fscanf(file,"%c",&s); //oppure s = getc(file);
        while (s==' ');
        return s;
}
```

Si fa notare che è stato definito un ciclo per la lettura dei singoli caratteri dal file in modo che vengano ignorati eventuali spazi posti tra i simboli delle produzioni.

Riconoscimento di simboli terminali

Quando un simbolo viene letto da un file, è possibile verificare se è un terminale, semplicemente controllando che esso sia compreso tra i caratteri ASCII '`a`' e '`z`' (oppure usando una opportuna funzione prevista dalle librerie del linguaggio C).

La funzione di riconoscimento acquisirà in input una variabile di tipo `Symbol` e restituirà un intero[1] che ne indicherà se si tratta di un terminale o meno. La dichiarazione e la definizione della funzione sono descritte di seguito:

```
int is_terminal(Symbol s)
{
    return (islower(s));
}
```

Riconoscimento di simboli non terminali

La funzione è analoga a quella per il riconoscimento dei simboli terminali, con la variante che la funzione restituisce valore vero se il simbolo è compreso tra i caratteri ASCII '`A`' e '`Z`', ovvero se è un carattere maiuscolo:

```
int is_nonterminal(Symbol s)
{
    return (isupper(s));
}
```

[1] Si ricorda che nel linguaggio C non esiste il tipo **`Boolean`** del Pascal. I valori logici sono rappresentati da numeri interi (per esempio, di tipo **`int`**), di modo che il valore 0 viene interpretato come FALSO, mentre tutti i possibili valori diversi da 0 sono interpretati come VERO.

Riconoscimento del separatore testa-corpo di una produzione

In una grammatica è necessario separare la testa e il corpo delle produzioni mediante un simbolo speciale che non sia né terminale né non terminale. Nell'analisi del problema è stato assunto che tale simbolo corrisponda al carattere ASCII '>'. Il riconoscimento di tale simbolo è immediato e si ottiene attraverso la seguente funzione:

```
19 int is_prodsym(Symbol s)
20 {
21     return (s == '>');
22 }
```

Riconoscimento del separatore tra produzioni

Le produzioni nel file di testo sono separate per mezzo di un "ritorno carrello", ossia un carattere ASCII particolare che non ha una propria rappresentazione grafica, ma la sua presenza è dovuta alla pressione del tasto "Invio" (↵) della tastiera. Nel linguaggio C, la costante che denota il ritorno a capo (detto anche "newline") è rappresentata con la scrittura `'\n'`[2] ed è equivalente al valore numerico 19.

Per il riconoscimento del separatore tra produzioni, la seguente funzione viene definita:

```
23 int is_prodsep(Symbol s)
24 {
25     return (s == '\n');
26 }
```

Se il simbolo contenuto in `s` non verifica nessuna delle funzioni di riconoscimento (`is_terminal`, `is_nonterminal`, `is_prodsym` e `is_prodsep`), allora si tratta di un simbolo estraneo alla definizione stabilita di grammatica. In questo caso, la procedura di caricamento della grammatica da file dovrebbe terminare con una segnalazione di errore.

Stampa di un simbolo

La stampa di un simbolo si ottiene semplicemente stampando il carattere ASCII che lo rappresenta. Per migliorare la leggibilità, è possibile far seguire la stampa del carattere da uno spazio:

```
27 void print_sym (Symbol s)
28 {
29      printf("%c ",s);
30 }
```

[2] L'uso dello "slash" ('\') serve ad evitare l'ambiguità del simbolo newline con la lettera 'n'. Ciò *non* implica che il newline è rappresentato da due caratteri ASCII.

Parole

Le produzioni di una grammatica sono definite da una testa e da un corpo. Entrambi questi elementi sono definiti come parole sull'alfabeto dei simboli terminali e non terminali. Si definisce pertanto il termine 'parola' in questo contesto come una sequenza di simboli terminali o non terminali.

Struttura dati

Una parola può essere rappresentata da diverse strutture dati, in funzione degli scopi e della complessità del programma in cui essa è utilizzata. In questo caso di studio, si rappresenta una parola come una sequenza di simboli di lunghezza variabile. Per semplicità, si assume che una parola non possa superare una determinata lunghezza massima e che la memorizzazione della parola avvenga in un array statico[3].

Si fissa la lunghezza massima di una parola a 100 simboli attraverso la definizione della seguente costante[4]:

```
31 #define MAX_WORD_LENGTH 100
```

La struttura dati che conterrà una parola è definita da un array di 100 simboli e da un campo che indica il numero di simboli effettivamente memorizzato nell'array[5]:

```
32 typedef struct
33 {
34         Symbol word [MAX_WORD_LENGTH];
35         unsigned length;
36 } Word;
```

Procedure e funzioni

L'aggiunta di simboli in una parola potrebbe avvenire semplicemente con istruzioni di assegnamento, ma si preferisce definire una procedura apposita.

```
37 void add_symbol (Word* w, Symbol s)
38 {
39      w->word[w->length++]=s;
40 }
```

Si noti l'utilizzo di un puntatore alla parola, visto che è necessaria una modifica alla parola stessa per via dell'aggiunta di un nuovo simbolo.

---

[3] Una struttura dati è statica se la sua occupazione di memoria è nota al momento della compilazione del programma. Le strutture statiche sono più semplici da usare ma possono determinare uno spreco di memoria o limitano la dimensione dei dati che si possono memorizzare.

[4] L'uso di costanti simboliche è utile per concentrare tali informazioni in un unico punto del programma (solitamente all'inizio), cosicché sia facile individuarle ed eventualmente modificarle. Inoltre utilizzando costanti simboliche si scrive il valore della costante una sola volta e lo si può usare quando necessario.

[5] La dichiarazione esplicita del tipo è **`unsigned int`**. Il linguaggio C consente, tuttavia, di omettere la parola chiave **`int`** in numerosi casi, tra cui nella dichiarazione delle variabili.

Per stampare una parola si può definire una funzione specifica che fa uso della procedura `print_sym` definita per la stampa dei singoli simboli:

```
41 void print_word (Word* w)
42 {
43       int i;
44       for (i=0; i < w->length; i++)
45           print_sym(w->word[i]);
46 }
```

Sebbene il parametro `w` non venga modificato all'interno della procedura (infatti, è un parametro di input) esso viene passato per riferimento (ossia, viene passato un puntatore ad una struttura di tipo `Word`). La scelta di passare un puntatore ad una struttura è dovuta esclusivamente al miglioramento dell'efficienza del programma. Infatti, passando un puntatore ad una struttura dati, l'occupazione di memoria richiesta per questa durante la chiamata alla procedura consiste solo nei byte necessari a contenere l'indirizzo della struttura, e non la struttura stessa che invece potrebbe occupare centinaia o migliaia di byte. In linea di principio, tuttavia, si potrebbe passare il parametro `w` per valore, poiché si tratta di un parametro di input e non di output, ma a discapito di un potenziale dispendio di memoria e di tempo[6].

Si noti nella funzione `add_symbol` e nella funzione `print_word` l'accesso ai membri di `w`. Poiché `w` è dichiarato come *puntatore* ad una struttura dati di tipo Word, l'accesso ai membri non avviene mediante l'operatore '`.`' ma attraverso l'operatore di accesso '`->`'. L'accesso all'array word, membro di `w`, avviene attraverso l'operatore di accesso agli elementi di un array '`[]`'. Si osservi che il primo elemento di un array ha indice 0. Di conseguenza, l'insieme degli indici per l'accesso ai simboli contenuti in word è {`0, 1, …, w->length-1`}. Per evitare che l'indice i non vada oltre questo range, la condizione di terminazione del ciclo **`for`** nella seconda funzione è di stretta disuguaglianza.

[6] Il tempo necessario a copiare l'intera struttura nel record di attivazione della procedura.

Produzioni

Struttura dati

Una produzione è definita da una coppia di parole: la testa (o parte sinistra) e il corpo (o parte destra). Essa può essere dunque semplicemente rappresentata da una struttura dati con due parole.

```
47 typedef struct
48 {
49         Word left;
50         Word right;
51 } Production;
```

Procedure e funzioni

La valorizzazione di una produzione può avvenire direttamente accedendo ai due membri `left` e `right`. Non vengono pertanto definite procedure specifiche. Si definisce una procedura per la visualizzazione di una produzione, realizzata nel seguente modo:

```
52 void print_production (Production* p)
53 {
54      print_word(&p->left);
55      printf (" --> ");
56      print_word(&p->right);
57 }
```

Si osservi che la funzione `print_word` richiede come parametro un puntatore ad un elemento di tipo `Word`. Di conseguenza, nell'uso di tale funzione è necessario passare l'indirizzo dei membri `left` e `right` della produzione passata (tramite puntatore) alla funzione `print_production`. Per ottenere l'indirizzo di tali membri si usa l'operatore '&'.

Grammatiche

Una grammatica viene rappresentata, per semplicità, esclusivamente come insieme di produzioni. Questa rappresentazione è giustificata dalle assunzioni semplificative dichiarate in fase di analisi del problema, grazie alle quali gli altri elementi definitori di una grammatica – l'alfabeto dei terminali, l'alfabeto dei non terminali e il simbolo di partenza – sono automaticamente determinati dall'elenco delle produzioni.

Struttura dati

Per rappresentare una grammatica, la soluzione più semplice consiste nel definire un array di produzioni. La dimensione di tale array può essere stabilita a priori, meglio se mediante una costante come ad esempio:

```
58 #define MAX_PRODUCTIONS 100
```

Poiché i file di testo contengono di norma un numero di produzioni inferiore a quello specificato da `MAX_PRODUCTIONS`, è necessario che un struttura dati contenente una grammatica contenga anche l'informazione relativa al numero di produzioni effettivamente memorizzato in essa. La struttura dati risultante è:

```
59 typedef struct
60 {
61         Production productions[MAX_PRODUCTIONS];
62         unsigned numprod;
63 } Grammar;
```

Il dato membro `numprod` è definito come **`unsigned`**. Questo tipo è sufficiente per memorizzare qualunque numero indicante il numero di produzioni in una grammatica.

Procedure e funzioni

Stampa di una grammatica

La procedura di stampa di una grammatica è semplice, poiché fa uso della procedura già precedentemente definita per la stampa delle produzioni:

```
64 void print_grammar(Grammar* g)
65 {
66      int i;
67
68      if (g == NULL)
69         printf ("Errore! Grammatica non valida! \n");
70      else
71      {
72         printf ("Num. di produzioni: %d\n", g->numprod);
73         for (i=0; i<g->numprod; i++)
74         {
75             print_production(&g->productions[i]);
76             printf ("\n");
77         }
78      }
79 }
```

La procedura di stampa richiede un puntatore ad una grammatica, anche se questa non viene modificata. Il passaggio di un puntatore consente di ottimizzare lo spazio di memoria, giacché il passaggio per valore di una grammatica richiederebbe la copia dell'intera struttura di una grammatica, che può occupare anche diverse centinaia di byte.

Innanzitutto, la procedura verifica se la grammatica passata in input è valida oppure non lo è. Per verificare se è valida, si verifica semplicemente che il puntatore passato non sia *nullo*, ossia non punti a nulla. Il test `g==NULL` determina l'esito di questa verifica[7].

Se il test è verificato, la procedura stampa dapprima il numero di produzioni e poi procede, una riga per volta, alla stampa delle singole produzioni utilizzando la funzione `print_production`.

[7] `NULL` non è una parola chiave, ma è una macro, che viene sostituita dal pre-processore dall'espressione `(void*) 0`, che indica, appunto, un puntatore nullo.

Caricamento di una grammatica da file

Questa procedura è la più importante e complessa del caso di studio. Essa ha lo scopo di leggere un file di testo e popolare una struttura di tipo `Grammar` con le produzioni definite nel file.

La dichiarazione della funzione è:

```
80 Grammar* load_grammar(FILE* file, Grammar* g)
```

La funzione prende in input un puntatore a `FILE` (che si assume aperto in precedenza), nonché un puntatore ad una struttura di tipo `Grammar` che conterrà la grammatica acquisita dal file. La funzione restituisce un puntatore a `Grammar`, che di fatto coinciderà con il valore del parametro `g`. Questa modalità è comune nella programmazione in C poiché consente di utilizzare la funzione sia come procedura, p.e.:

```
load_grammar(my_f, my_g);
```

oppure come funzione, come per esempio

```
my_g = load_grammar(my_f, my_g);
```

o

```
print_grammar(load_grammar(my_f, my_g));
```

Per la scansione del file, si dichiara una variabile locale di tipo `Symbol` che conterrà l'ultimo simbolo letto dal file. La dichiarazione è semplicemente:

```
81 Symbol s;
```

Durante l'acquisizione di una produzione, si può immaginare che il processo di scansione può trovarsi in uno di quattro stati:

- **START**: Il processo di scansione comincia ad acquisire una produzione;
- **LEFT**: Il processo di scansione è in fase di acquisizione della parte sinistra di una produzione;
- **RIGHT**: Il processo di scansione è in fase di acquisizione della parte destra di una produzione;
- **ERROR**: Il processo di scansione deve essere arrestato perché il file non contiene una produzione scritta correttamente

Al fine di rappresentare questi stati, si definisce un tipo enumerativo, nel seguente modo:

```
82 enum States {START,LEFT,RIGHT,ERROR};
```

Successivamente si può dichiarare una variabile di questo tipo, che si può subito inizializzare a `START` giacché all'inizio della scansione si comincia ad acquisire la prima produzione:

```
83 enum States current_state = START;
```

Prima di procedere alla definizione del corpo della funzione, è opportuno dichiarare una variabile che conterrà il puntatore alla produzione in fase di acquisizione. Tale variabile è dichiarata come:

```
84 Production* p;
```

La funzione dapprima inizializza la grammatica, imponendo che il numero di produzioni sia 0. L'istruzione necessaria è:

```
85 g->numprod = 0;
```

Successivamente si comincia il processo di scansione. Questo è un processo iterativo, che si ripete ogni volta che viene letto un nuovo carattere dal file. Esso dunque termina quando è finito il file oppure quando si verifica una condizione di errore. Per verificare che un file è terminato, si può utilizzare la funzione `feof(FILE*)`, che restituisce valore vero se il processo di scansione tenta di leggere oltre la fine del file (eof = "end of file"). Lo stato di errore si verifica con un test sulla variabile di stato. Il ciclo di iterazione è definito dunque da:

```
86 while (current_state != ERROR && !feof(file))
```

All'interno del ciclo, la prima operazione da effettuare è la lettura di un simbolo dal file, utilizzando la funzione `read_sym` precedentemente definita:

```
87 s = read_sym(file);
```

Per ragioni tecniche, è opportuno verificare di nuovo che la condizione di fine file non sia stata raggiunta[8]. Se viene raggiunta, occorre terminare il ciclo. Per realizzare questa verifica (e la relativa azione), si introduce l'istruzione:

```
88 if (feof(file)) break;
```

A questo punto, occorre attuare il processo di riconoscimento del carattere e della relativa azione da intraprendere al fine della ricostruzione della produzione in fase di acquisizione. Questo processo dipende dallo stato corrente, pertanto è opportuno definire una selezione di casi mediante l'operatore di selezione multipla:

```
89 switch(current_state)
```

[8] Questa doppia verifica è necessaria perché alcuni editor di testo inseriscono un carattere ASCII speciale (il carattere numero 27, non stampabile, denominato ESC) alla fine del file.

Sulla base del valore di `current_state`, si intraprenderanno diverse azioni.

Il primo caso si verifica quando `current_state` è valorizzato a `START`. In questo caso, se il simbolo letto è un terminale o un non terminale, allora occorrerà passare allo stato `LEFT` (scansione della parte sinistra) e inserire il simbolo come primo nella testa della produzione in fase di acquisizione. Se il simbolo è un newline (separatore *tra* produzioni), allora lo stato corrente torna a `START` perché occorre prepararsi alla scansione di una nuova produzione. Se il simbolo non è terminale o non-terminale, si verifica una condizione di errore. Questa azione si realizza con il seguente codice:

```
90  case START:
91        if (is_terminal(s) || is_nonterminal(s))
92        {
93            current_state = LEFT;
94            p=add_new_production(g);
95            add_symbol(&p->left,s);
96        }
97        else if (isprodsep(s))
98        {
99            current_state = START;
100       }
101       else
102           current_state = ERROR;
103       break;
```

L'istruzione `p=add_new_production(g)` permette di aggiungere una nuova produzione alla grammatica. Si osservi la composizione della funzione:

```
104 Production* add_new_production(Grammar *g)
105 {
106   Production* p;
107   p = &(g->productions[g->numprod++]);
108   p->left.length = 0;
109
110   return p;
111 }
```

L'istruzione composta `p = &(g->productions[g->numprod++])` valorizza la variabile `p` (di tipo puntatore a `Production`) all'indirizzo di una produzione memorizzata nella grammatica `g`. Quale produzione? Quella nell'array `g->productions` all'indice `g->numprod`. Questo indice viene (dopo l'operazione di accesso all'array) incrementato di una unità per indicare che la grammatica si sta arricchendo di una nuova produzione.

Dopo aver acquisito tale indirizzo, il contenuto di `p` (e quindi dell'elemento dell'array di produzioni che esso referenzia) viene opportunamente modificato. La lunghezza della parte sinistra viene inizialmente azzerata (`p->left.length=0`) e successivamente viene memorizzato nel primo elemento della parte sinistra il simbolo letto dal file tramite la funzione `add_symbol`.

Nel caso in cui il processo di acquisizione sia nel mezzo della lettura della parte sinistra di una produzione, lo stato corrente del processo è impostato a `LEFT`. In tale stato, ad una lettura di un simbolo occorre verificare se:

- Il simbolo è un terminale o un non-terminale. In tal caso, è sufficiente memorizzare il simbolo nella parte sinistra della produzione in fase di acquisizione e continuare il processo;
- Il simbolo è un separatore testa-corpo. In questo caso, lo stato corrente deve passare a RIGHT, il simbolo letto deve essere scartato (perché si tratta del separatore, quindi non è necessario memorizzarlo) e il processo può andare avanti;
- In altri casi (per esempio, se il simbolo letto è una newline), si verifica una condizione di errore

Il codice che realizza questa sequenza di azioni condizionali è il seguente:

```
112 case LEFT:
113      if (is_terminal(s) || is_nonterminal(s))
114      {
115         current_state = LEFT;
116         add_symbol(&p->left,s);
117      }
118      else if (is_prodsym(s))
119      {
120         current_state = RIGHT;
121         p->right.length = 0;
122      }
123      else
124         current_state = ERROR;
125      break;
```

Il caso in cui lo stato corrente sia `RIGHT` indica che il processo di acquisizione è nel mezzo della lettura della parte destra di una produzione. Il comportamento della funzione in questo stato è simile a quello dello stato `LEFT`, con la variante che se si incontra il simbolo di separatore di produzione, si passa ad uno stato di errore (una produzione, infatti, non può essere scritta come p.e. `Aa>bB>c`). Se invece il simbolo letto è una newline (separatore *tra* produzioni), allora lo stato corrente torna a `START` perché occorre prepararsi alla scansione di una nuova produzione. Il seguente codice realizza il comportamento della funzione nello stato `RIGHT`:

```
126 case RIGHT:
127      if (is_terminal(s) || is_nonterminal(s))
128      {
129         current_state = RIGHT;
130         add_symbol(&p->right,s);
131      }
132      else if (is_prodsep(s))
133      {
134          current_state = START;
135      }
```

```
136       else
137           current_state = ERROR;
138       break;
```

Il processo iterativo di acquisizione delle produzioni termina quando si raggiunge la fine del file oppure si verifica una condizione di errore. Questa condizione di errore si verifica quando lo stato corrente è stato impostato su `ERROR`, oppure si è raggiunta la fine del file mentre si stava acquisendo la parte sinistra di una produzione. Viceversa, se lo stato corrente è impostato su `START` oppure su `RIGHT` al termine del ciclo, è garantito che la grammatica acquisita è valida. In questo caso, la funzione può restituire il puntatore alla grammatica valorizzata. In condizioni di errore, invece, è utile restituire un puntatore nullo, cosicché la procedura che usa `load_grammar` potrà verificare se il processo di acquisizione è andato a buon fine oppure no. La parte rimanente della funzione è dunque così codificata:

```
139 if (current_state == START || current_state == RIGHT)
140    return g;
141 else
142    return NULL;
```

Il programma principale

Il *main program* dovrà essere responsabile di aprire il file, leggere la grammatica, stamparla e chiudere il file. Per specificare il file, si dichiarano i parametri della funzione `main` nel seguente modo[9] :

```
143 int main(int argc, char *argv[])
```

Tali parametri sono utili per utilizzare il programma di lettura delle grammatiche a linea di commando. Per esempio, se il codice sorgente viene tradotto nel programma eseguibile `GrammarReader.exe`, si potrà specificare il nome del file contenente la grammatica con il seguente comando

`GrammarReader` *`nomefile`*

per esempio:

```
GrammarReader mygrammar.txt
```

Per acquisire il nome del file specificato nella linea di comando, la funzione `main` dovrà avere i parametri specificati nel modo precedentemente descritto.

Il significato dei parametri è il seguente:

– `argc` specifica il numero di parametri specificati nella linea di comando, incluso il nome del programma eseguibile;

[9] Si tratta di una modalità standard: o non si specificano i parametri per la funzione `main`, oppure si specificano nel modo illustrato.

– `argv` è un puntatore ad un array di stringhe. Ciascun elemento dell'array contiene un parametro specificato nella linea di comando, incluso il nome del file eseguibile. Cosicché, per l'esempio precedente, `argv[0]` è "`GrammarReader`", mentre `argv[1]` è "`mygrammar.txt`"

Nel programma principale si potrà dunque dichiarare una variabile locale, di tipo stringa (**`char*`**) valorizzata ad `argv[1]`, così da contenere il nome del file da leggere. Se per errore il nome del file non è stato specificato, la variabile locale avrà valore nullo.

```
144 char* filename = argv[1];
```

Successivamente, si potrà dichiarare una variabile di tipo `FILE*` che conterrà le informazioni sul flusso aperto sul file specificato.

```
145 FILE* gram_file;
```

Infine, si dichiara una variabile di tipo `Grammar` che sarà valorizzata dalla procedura di acquisizione della grammatica da file.

```
146 Grammar grammar;
```

Il programma principale dapprima verifica se l'utente ha correttamente specificato un nome di file nella linea di comando. In caso negativo, il programma termina con un errore:

```
147 if (filename == 0)
148   {
149      printf("nome file non specificato \n");
150      return -1;
151   }
```

Se il test è positivo, si tenta di aprire un flusso sul file specificato. Se l'apertura del flusso non ha successo (p.e. se si specifica un nome di file inesistente), il programma terminerà con un altro messaggio di errore:

```
152   gram_file = fopen(filename,"r");
153   if (gram_file == NULL)
154   {
155      printf("nome di file errato\n");
156      return -1;
157   }
```

Se anche questo test è positivo (e dunque il flusso è correttamente aperto), si può procedere all'acquisizione della grammatica e alla sua stampa, mediante l'istruzione composta:

```
158 print_grammar(load_grammar(gram_file,&grammar));
```

Infine, si può chiudere il flusso aperto, con l'istruzione:

```
159 fclose(gram_file);
```

Il programma può ritenersi terminato. Per migliorarne l'interattività, si può chiedere all'utente di premere un tasto prima di chiudere la finestra del programma:

```
160   system("PAUSE");
161   return 0;
```