PROGETTO COMPILATORI

Corsa ciclisti

Dario Zappata: 0699471
Francesco La Rosa: 0696471

Analisi input

Il file di input segue una forma rigida in quanto è suddiviso in tre sezioni separate rispettivamente dalle stringhe “%%” e “$$”.
La prima sezione contiene una data che indicherà una tappa specifica tra quelle previste dalla gara.
Nella seconda sezione troviamo l’elenco dei partecipanti iscritti alla gara separati da “&&&”.
Mentre nella terza sezione saranno presenti le descrizioni di ciascuna tappa.

Analisi Lessicale

La prima fase prevede l’analisi lessicale, effettuata nel nostro caso con flex, in cui lo scanner scandisce la sequenza di caratteri del file di input raggruppandoli in lessemi e producendo una sequenza di token che successivamente verranno passati al parser.

La parte delle dichiarazioni contiene:

- le **librerie** tra cui l’header del parser necessario per la comunicazione tra i due analizzatori;
- le **definizioni regolari** che ricorrono nell’input;
- le **start conditions** utilizzate dall’analizzatore lessicale per risolvere le ambiguità date da una stringa che potrebbe avere un match con due espressioni regolari diverse.

```
%{
#include <string.h>
#include<stdlib.h>
#include "parser.tab.h"
%}
%option noyywrap
dataClassifica ([0][1-9]|[1-2][0-9]|([3][0-1]))[/](([0][1-9])|([1][0-2]))[/]([0-9]{2})
codiceAtleta [A-Z]{2}[0-9]{4}
nomeCogn [A-Z][a-z]*([ ][A-Z][a-z]*)*
squadraAtleta ([a-z]+[ ]?)+
nPettorale [1-9]|[1-9][0-9]|[1-9][0-9][0-9]
data ([0][1-9]|[1-2][0-9]|([3][0-1]))[/](([0][1-9])|([1][0-2]))
citta [A-Z][a-z]*([ ][A-Z]?[a-z]*)*
tempo [0-9]([:])[0-5][0-9]([:])[0-5][0-9]
%s condNome condCitta
```

Nella parte delle regole di traduzione abbiamo inserito una sequenza di regole composta dal pattern e il codice C da eseguire in corrispondenza di un match:

```
[ \t\n]+ ;
"Data:"                     {return(DATA);}
{dataClassifica}            {yylval.string= strdup(yytext); return(DATA_T);}
"%%"                        {return(SEP_SEZ_1);}
"Codice :"                  {return(CODICE);}
{codiceAtleta}              {yylval.string= strdup(yytext); return(CODICE_T);}
"Cognome e nome:"           {BEGIN(condNome); return(NOME_COGN);}
"Squadra:"                  {return(SQUADRA);}
{squadraAtleta}             {yylval.string= strdup(yytext); return(SQUADRA_T);}
"Pettorale:"                {return (PETTORALE);}
{nPettorale}                {yylval.intero= atoi(yytext); return(PETTORALE_T);}
"&&&"                       {return(SEP_ATL);}
"$$"                        {BEGIN(condCitta);return(SEP_SEZ_2);}
{data}                      {yylval.string= strdup(yytext); return(DATA_TAPPA_T);}
"-"                         {return(TRATTINO);}
"("                         {return(APRI_P);}
")"                         {return(CHIUDI_P);}
":"                         {return(DUE_PUNTI);}
"->"                        {return(FRECCIA);}
{tempo}                     {
                                char *s= strdup(yytext);
                                int h,m1,m2,s1,s2;
                                h= s[0]-'0';
                                m1=s[2]-'0';
                                m2=s[3]-'0';
                                s1=s[5]-'0';
                                s2=s[6]-'0';
                                yylval.intero= (h*3600+((m1*10+m2)*60)+(s1*10+s2));
                                return(TEMPO);
                            }
. {
        fprintf(stderr,"Rilevati caratteri non ammessi: %s \n",strdup(yytext));
        exit(1);
}

<condNome>{nomeCogn} { yylval.string=strdup(yytext); return(NOME_COGN_T);}
<condCitta>{citta}   { yylval.string=strdup(yytext); return(CITTA);}
```

Nel caso in cui vengono riconosciuti i token **{dataClassifica} {codiceAtleta} {squadraAtleta} {nPettorale} {nomeCogn} {citta} {data} {tempo}** attribuiamo alla variabile yylval la conversione in intero o stringa del lessema riconosciuto (yytext).

Inoltre in corrispondenza del token **{tempo}** abbiamo effettuato una conversione dal formato hh:mm:ss in secondi per facilitare i calcoli.

Analisi Sintattica

Nel file dato in input al generatore automatico di parser si definisce la grammatica per il linguaggio di input e si procede all'analisi della sintassi, e conseguentemente della semantica, sulla base dei token restituiti dallo scanner.

Nella parte delle definizioni del parser abbiamo incluso le librerie utili per il corretto funzionamento e in particolare la libreria **symb.tab.h** per accedere alle funzioni necessarie per operare sulla tabella dei simboli.

Sono presenti, inoltre, le definizioni di alcune variabili globali utilizzate per l'attribuzione del significato semantico in corrispondenza delle regole di produzione.

Il costrutto **%union** è stato definito per specificare la collezione di tipi provenienti dallo scanner e scegliere di volta in volta il tipo più utile per i token.

```
%{
    #include <stdio.h>
    #include <stdbool.h>
    #include <stdlib.h>
    #include <string.h>
    #include "symb.tab.h"
    int yylex();
    void yyerror (char const *);
    bool controlloData(char* dataClassifica, char* dataTappa);

    char *dataClassifica, *dataTappa, *nome;
    int nPettorale;
%}

%union {
    char *string;
    int intero;
}
```

Successivamente abbiamo elencato tutti i token passati dallo scanner, definito l'assioma e tutte le regole di produzione che definisco la grammatica utilizzata per l'analisi sintattica del problema in esame.

```
%token <string> DATA_T CODICE_T SQUADRA_T DATA_TAPPA_T NOME_COGN_T CITTA
%token <intero> PETTORALE_T TEMPO
%token DATA SEP_SEZ_1 CODICE NOME_COGN SQUADRA PETTORALE SEP_ATL SEP_SEZ_2
TRATTINO APRI_P CHIUDI_P DUE_PUNTI FRECCIA

%start s
```

Di seguito le regola di produzione che delimita le tre sezioni in cui è suddiviso l'input:

```
s: data SEP_SEZ_1 elenco_ciclisti SEP_SEZ_2 lista_tappe
```

Il simbolo non terminale data produce la stringa **"Data:"** seguito da **DATA_T** che contiene la data della tappa di cui vogliamo conoscere la classifica nel formato gg/mm/aa. Tale informazione verrà memorizzata nella variabile globale **dataClassifica**:

```
data: DATA DATA_T {dataClassifica= $2;}
```

La seconda sezione contiene l'elenco dei partecipanti iscritti alla competizione. A tal fine, la regola di produzione **elenco_ciclisti** genera un elenco di ciclisti in cui ciascun **ciclista** è caratterizzato da un **codice**, dal **nome** e **cognome**, dalla **squadra** e dal **numero di pettorale**.

```
elenco_ciclisti: elenco_ciclisti ciclista
        | ciclista;

ciclista: codice nome_cogn squadra n_pettorale SEP_ATL {
      insertCiclista(nPettorale,nome);
      resizeHashTable();}
        ;

codice: CODICE CODICE_T
      ;

nome_cogn: NOME_COGN NOME_COGN_T {nome= $2;}
         ;

squadra: SQUADRA SQUADRA_T
       ;

n_pettorale: PETTORALE PETTORALE_T {nPettorale= $2;}
           ;
```

La terza sezione contiene l'elenco delle tappe previste dalla competizione. A tal fine la regola di produzione **lista_tappe** genera l'elenco delle tappe ciascuna delle quali caratterizzata dalla **data** in cui si è svolta, le **città** di partenza e di arrivo ed infine dalla **classifica** per quella tappa.

```
lista_tappe: lista_tappe tappa
           | tappa ;

tappa: competizione TRATTINO classifica ;

competizione: DATA_TAPPA_T TRATTINO CITTA TRATTINO CITTA {dataTappa= $1;} ;

classifica: classifica FRECCIA tempo
          | tempo ;
```

La classifica di una tappa è caratterizzata dall'elenco dei tempi. La regola di produzione **tempo** tiene traccia del tempo impiegato da ciascun ciclista identificato dal suo **numero di pettorale**. Se la data della tappa che stiamo analizzando coincide con la data specificata alla prima riga dell'input, inseriamo il ciclista in una coda per tenere traccia della classifica della tappa. Mentre nel caso in cui la data della tappa è precedente o uguale alla data specificata alla prima riga dell'input calcoliamo la classifica parziale in base ai tempi impiegati dai ciclisti.

```
tempo: APRI_P PETTORALE_T DUE_PUNTI TEMPO CHIUDI_P {
      if((strncmp(dataClassifica, dataTappa, 5))==0){
                  insertTappa($2,$4);}
      if(controlloData(dataClassifica,dataTappa)){
                  impostaClassificaParziale($2,$4);}};
```

Tabella dei simboli

Per la gestione della tabella dei simboli abbiamo utilizzato le seguenti strutture dati:

- **Ciclista**: è una struct contenente le informazioni di un ciclista;
- **hashTable**: tabella hash contenente le informazioni dei ciclisti in cui le collisioni sono gestite mediante liste;
- **classificaTappa**: è una coda utilizzata per tenere traccia dell'ordine di arrivo dei ciclisti nella tappa specificata;
- **classificaParziale**: è una lista ordinata per i tempi parziali dei ciclisti per tutte le tappe fino alla data specificata

```
typedef struct ciclista{
    char* nome;
    int pettorale;
    int tempo;
    int tempoParziale;
    struct ciclista *next;
}Ciclista;

Ciclista** hashTable;

typedef struct queue{
    Ciclista* testa;
    Ciclista* coda;
}Queue;

Queue* classificaTappa;

Ciclista* classificaParziale;
```

Inizialmente, abbiamo dichiarato quattro variabili globali che ci saranno utili in alcune funzioni. In particolare, alcune di esse sono utilizzate per il ridimensionamento della tabella hash e le altre sono di supporto per il calcolo della classifica parziale.

```
int SIZE= 3;
double threshold= 0.75;
int n= 0;
int posizione= 1, ciclistiInTappa=1;
```

Successivamente abbiamo inserito le funzioni principali per la gestione delle interazioni tra parser e symbol table.

La funzione **start()** inizializza le strutture che ci serviranno nelle successive funzioni.

```
void start(){
    classificaTappa=(Queue*) malloc(sizeof(Queue));
    classificaTappa->testa= NULL;
    classificaTappa->coda= NULL;

    classificaParziale= NULL;

    hashTable= malloc(SIZE*sizeof(*hashTable));
    for(int i=0; i<SIZE; i++)
        hashTable[i]=NULL;
}
```

La funzione hash che abbiamo scelto è la divisione con modulo, con cui calcoliamo la cella in cui inserire il ciclista nella tabella hash.

```
int hash(int pettorale){
    return pettorale%SIZE;
}
```

La funzione **insertCiclista()**, mediante la funzione hash descritta precedentemente, aggiunge il ciclista nella tabella hash.

```
void insertCiclista(int pettorale, char* nome){
    Ciclista* cicl;
    if((cicl=ricerca(pettorale))==NULL){
        cicl=(Ciclista*)malloc(sizeof(*cicl));
        cicl->pettorale= pettorale;
        cicl->nome= nome;
        cicl->tempo= 0;
        cicl->tempoParziale= 0;
        cicl->next= hashTable[hash(pettorale)];
        hashTable[hash(pettorale)]= cicl;
        n++;
    }
    else{
        fprintf(stderr,"Sono presenti due ciclisti con il pettorale:
%d\n",pettorale);
        exit(1);
    }
}
```

La funzione **ricerca()** verifica se un ciclista è presente nell'hashTable.

```
Ciclista* ricerca(int pettorale){
    Ciclista* ciclista;
    for(ciclista=hashTable[hash(pettorale)]; ciclista!=NULL; ciclista=ciclista->next){
        if(ciclista->pettorale==pettorale)
            return ciclista;
    }
    return NULL;
}
```

La funzione **resizeHashTable()** effettua il ridimensionamento dell'hashTable mediante la tecnica del doubling in cui per ciascuno dei ciclisti presenti nell'hashTable ricalcola la funzione hash e li dispone nelle rispettive nuove celle.

```
void resizeHashTable(){
    double loadFactor= (double)n/(double)SIZE;
    if(loadFactor >= threshold){
        Ciclista** oldHashTable=hashTable;
        n=0;
        SIZE*=2;
        hashTable=NULL;
        hashTable= malloc(SIZE*sizeof(*hashTable));
        for(int i=0; i<SIZE; i++)
            hashTable[i]=NULL;
        for(int i=0; i<SIZE/2; i++){
            Ciclista* ciclista= oldHashTable[i];
            while(ciclista!=NULL){
                insertCiclista(ciclista->pettorale, ciclista->nome);
                ciclista=ciclista->next;
            }
        }
        free(oldHashTable);}}
```

La funzione **insertTappa()** inserisce i ciclisti nella coda **classificaTappa** in base all'ordine di arrivo della tappa specificata

```
int insertTappa(int pettorale, int tempo){
    Ciclista* ciclista= hashTable[hash(pettorale)];
    while((ciclista!=NULL) && ((ciclista->pettorale)!=pettorale)){
        ciclista=ciclista->next;}

    if(ciclista==NULL){
        fprintf(stderr,"Il ciclista %d non e' presente nell'elenco dei
                partecipanti\n", pettorale);
        exit(1);}

    Ciclista* temp=(Ciclista*) malloc(sizeof(Ciclista));
    temp->nome= ciclista->nome;
    temp->pettorale= pettorale;
    temp->tempo= tempo;
    temp->next=NULL;
    if(classificaTappa->coda==NULL){
        classificaTappa->coda= temp;
        classificaTappa->testa= temp;
    }else{
        classificaTappa->coda->next= temp;
        classificaTappa->coda= temp;}}
```

La funzione **impostaClassificaParziale()** calcola i tempi parziali con gli eventuali bonus in base alla posizione di arrivo.

```
void impostaClassificaParziale(int pettorale, int tempo){
    Ciclista* ciclista= hashTable[hash(pettorale)];
    while((ciclista!=NULL) && ((ciclista->pettorale)!=pettorale)){
        ciclista=ciclista->next;}

    if(ciclista==NULL){
        fprintf(stderr,"Il ciclista %d non e' presente nell'elenco dei
                partecipanti\n",pettorale);
        exit(1);}
    /*se ciclistiInTappa è maggiore di n significa che stiamo iniziando ad
      analizzare la tappa successiva e quindi resetta le posizioni per il
      calcolo dei bonus */
    if(ciclistiInTappa>n){
        ciclistiInTappa=1;
        posizione=1;}

    int tempoBonus= 0;
    if(posizione==1)
        tempoBonus= 60;
    else if(posizione==2)
        tempoBonus= 30;
    else if(posizione==3)
        tempoBonus= 10;
    ciclista->tempoParziale+=(tempo-tempoBonus);

    ciclistiInTappa++;
    posizione++;}
```

La funzione **printTappa()** è utilizzata per la stampa della classifica della tappa specificata. In questa fase è utilizzata la coda **classificaTappa**.

```
void printTappa(char* dataClassifica){
    Ciclista* ciclista= classificaTappa->testa;
    int posto=1;
    printf("\nCLASSIFICA TAPPA %s\n",dataClassifica);
    while(ciclista!=NULL){
        printf("   %d.%25s   ",posto++,ciclista->nome);
        //conversione da secondi a hh:mm:ss
        int h= (ciclista->tempo)/3600;
        int m= ((ciclista->tempo)-(3600*h))/60;
        int s= ((ciclista->tempo)-(3600*h)- (m*60));
        printf("%.2d:%.2d:%.2d\n",h,m,s);
        ciclista=ciclista->next;}}
```

La funzione **printClassificaParziale()** è utilizzata per la stampa della classifica parziale fino alla data specificata, scorrendo gli elementi nella coda **classificaTappa** per creare la lista ordinata per tempi parziali.

```
void printClassificaParziale(){
    Ciclista* ciclista= classificaTappa->testa;
    int posto=1;
    while(ciclista!=NULL){
        Ciclista* cicl= hashTable[hash(ciclista->pettorale)];
        while((cicl->pettorale != ciclista->pettorale)){
            cicl= cicl->next;}
        ordinaTempi(cicl->nome, cicl->tempoParziale);
        ciclista= ciclista->next;}

    Ciclista* listaOrdinata= classificaParziale;
    printf("\nCLASSIFICA PARZIALE\n");
    while(listaOrdinata!=NULL){
        printf("   %d.%25s   ",posto++,listaOrdinata->nome);
        //conversione da secondi a hh:mm:ss
        int h= (listaOrdinata->tempoParziale)/3600;
        int m= ((listaOrdinata->tempoParziale)-(3600*h))/60;
        int s= ((listaOrdinata->tempoParziale)-(3600*h)- (m*60));
        printf("%.2d:%.2d:%.2d\n",h,m,s);
        listaOrdinata= listaOrdinata->next;}}
```

La funzione **ordinaTempi(),** invocata dalla funzione precedente, è utilizzata per l'inserimento ordinato nella lista **classificaParziale** in base ai tempi parziali dei ciclisti.

```
void ordinaTempi(char* nome,int tempoParziale){
    Ciclista* temp =(Ciclista*) malloc(sizeof(Ciclista));
    temp->nome = nome;
    temp->tempoParziale= tempoParziale;
    temp->next = NULL;

    Ciclista *prev = NULL,*current = classificaParziale;
    //ricerca della posizione corretta nella lista
    while((current!=NULL) && ((temp->tempoParziale)>(current->tempoParziale))){
        prev= current;
        current= current->next;}

    if(prev== NULL){
        temp->next= classificaParziale;
        classificaParziale= temp;
    }else{
        prev->next= temp;
        temp->next= current;
    }}
```

Inoltre, al fine di liberare la memoria allocata per tutte le strutture utilizzate abbiamo utilizzato la funzione **freeAll()**.

Gestione degli errori

Gli **errori lessicali** vengono rilevati durante l'analisi lessicale quando si intercettano caratteri non ammessi. L'istruzione che li cattura è la seguente:

```
. {   fprintf(stderr,"Rilevati caratteri non ammessi: %s \n",strdup(yytext));
      exit(1);
  }
```

Gli **errori sintattici** sono rilevati dal parser ogni volta che la grammatica non viene rispettata. In questo caso la funzione **yyerror** stampa un messaggio di errore:

```
void yyerror (char const *s) {
    fprintf(stderr, "Sono presenti degli errori sintattici\n%s\n", s);
}
```

Gli **errori semantici** sono rilevati durante l'esecuzione di alcune funzioni presenti in symbol.table.c descritte precedentemente.

Esempio esecuzione: input

Data: 23/05/20
%%
Codice : PQ3274
Cognome e nome: Giovanni Amato
Squadra: libertas ct
Pettorale: 215
&&&
Codice : PM4285
Cognome e nome: Pietro Magri
Squadra: asd ciclismo pa
Pettorale: 357
&&&
Codice : AP8242
Cognome e nome: Marco Bonanno
Squadra: libertas ct
Pettorale: 818
&&&
Codice : AZ6813
Cognome e nome: Massimo Bonelli
Squadra: palermo cicismo
Pettorale: 203
&&&
Codice : BY9532
Cognome e nome: Antonio Cusimano
Squadra: palermo ciclismo
Pettorale: 613
&&&
Codice : CA4794
Cognome e nome: Paolo Di Blasi
Squadra: sporting club
Pettorale: 418
&&&
Codice : PN1137
Cognome e nome: Manfredi Terranova
Squadra: asd cefau
Pettorale: 206
&&&
$$
22/05 - Palermo - Cefalu - (203:2:42:30) -> (215:2:42:53) -> (206: 2:43:15) -> (613: 2:43:16) -> (418: 2:43:21) -> (357: 2:44:02) -> (818: 2:44:50)
23/05 - Cefalu- Messina - (215: 3:55:34) -> (613: 3:55:35) -> (206: 3:55:50) -> (418: 3:55:52) -> (203: 3:56:48) -> (818: 3:57:10) -> (357:3:58:30)

Esempio esecuzione: output

CLASSIFICA TAPPA 23/05/20

1. Giovanni Amato 03:55:34
2. Antonio Cusimano 03:55:35
3. Manfredi Terranova 03:55:50
4. Paolo Di Blasi 03:55:52
5. Massimo Bonelli 03:56:48
6. Marco Bonanno 03:57:10
7. Pietro Magri 03:58:30

CLASSIFICA PARZIALE

1. Giovanni Amato 06:36:57
2. Massimo Bonelli 06:38:18
3. Antonio Cusimano 06:38:21
4. Manfredi Terranova 06:38:45
5. Paolo Di Blasi 06:39:13
6. Marco Bonanno 06:42:00
7. Pietro Magri 06:42:32