Relazione Compilatori

Progetto “Tournèe”

Musotto Daniela (0678679) - Martinello Pierfrancesco (0679802)

PANORAMICA

Lo scopo del programma “Tournèe” è quello di analizzare le diverse tappe della tournèe di un gruppo musicale e calcolare l’importo guadagnato dalla vendita dei biglietti e quanto viene conferito in beneficenza per ogni tappa e in totale.

Viene fornito al programma un file di testo in input contenente i dettagli delle varie tappe, definite con:

- il nome della città;
- la location del concerto;
- la data;
- i vari tipi di biglietto, per i quali vengono descritti i numeri di biglietti staccati per una tale categoria;

In una seconda sezione dell’input viene descritto il prezzo per tipo di biglietto. Infine, una sezione finale conterrà gli sconti per ogni categoria.

Se l’input è corretto, il programma restituirà in output una prima sezione contenente il ricavato e il rispettivo importo in beneficenza per ogni tappa in input, e una seconda sezione con l’importo totale ottenuto dalla vendita dei biglietti di tutte le tappe e il totale dato in beneficenza.

L’applicazione è realizzata utilizzando Flex, un generatore automatico di analizzatori lessicali, e Bison, un generatore automatico di analizzatori sintattici.

Vengono fornite due diverse cartelle “Pre-Ver 3.6” e “Ver 3.6” . Entrambe hanno lo stesso codice, ma la differenza è nell’istruzione che specifica una descrizione verbosa degli errori in output.

Il codice in Ver 3.6 è stato analizzato in sviluppo con Bison versione 3.7.5 la cui specifica è $define parse.error verbose

Il codice in Pre-Ver 3.6 è stato adatto a versioni di Bison inferiori a 3.6 la cui specifica è %error-verbose

Fase 1 - Input

L'input è suddiviso in tre sezioni fondamentali: **lista delle tappe**, **sezione dei prezz**i e **sezione degli sconti**. Tali sezioni sono divise mediante operatori simbolici.

Lista delle tappe

Ogni tappa del tour è definita come segue:

```
Città: <nome_citta<;
Location: <nome_location>;
Data: <data_tappa1>;
Biglietti cat.<tipo biglietto>: (elenco numero biglietti per categoria);
...
```

Dove i biglietti possono essere di vario tipo (indicato con una lettera maiuscola).
//Ogni categoria ha diritto a uno sconto specifico sul prezzo del biglietto. Le categorie sono: Ordinario, Musicisti, Junior, Senior, Disabili, Autorità.
Nell'elenco, la categoria è indicata dalla sua iniziale seguita da ":" e dal numero (intero) di biglietti staccati in quella categoria, separati da ";". Per esempio:

```
<nominativo categoria: <numero di biglietti staccati>; <nominativo categoria: <numero di biglietti staccati>; ...
```

La fine dei dettagli di una tappa è indicata con una linea di 20 asterischi "`**********`".

La fine della sezione della lista delle tappe è indicata con tre simboli di dollaro, "`$$$`".

Sezione dei prezzi

La sezione dei prezzi contiene la lista dei prezzi associati ad una determinata tipologia di biglietto, nel seguente formato;

```
Prezzo<tipo biglietto> --> <prezzo tipo>.
...
```

La fine della sezione dei prezzi è indicata con con una linea con 20 simboli +, "`++++++++++++++++++++`".

Sezione degli sconti

Nella terza e ultima sezione, sono indicati gli sconti relativi ad ogni categoria. Vengono indicati come una serie di righe definite dal nome della categoria di riduzione, seguito da ":" e la percentuale di sconto:

```
<Nome_categoria>: <perc_sconto>%
```

Fase 2 - Analizzatore Lessicale

L'analizzatore lessicale è realizzato in Flex, e prevede la tokenizzazione dei seguenti elementi:

Regex	Token
";"	DIV
":"	OP
"("	OPEN
")"	CLOSED
"%"	PERC
"EURO."	VALUE
"-->"	ARR
"-"	DATEDIV
"Città:"	NOMEC_PREF
"Location:",	NOMEL_PREF
"Data:"	DATE_PREF
"Biglietti cat."	BIGL_PREF
"Prezzo"	PREZ_PREF
"********************"	SEP
"$$$\nPREZZI BIGLIETTI"	SEP1
"++++++++++++++++++++\nSCONTI"	SEP2
"[\n\t]"	

Gli unici token i cui attributi vengono passati all'analizzatore sintattico tramite la variabile Flex `yylval` sono: `NUM, NOMEC, NOMEL, NOMET`:

- `NUM` deriva dalla regex ***number*** `{digit}+` ;
- `NOMEC` e `NOMEL` derivano rispettivamente dalle regex
    - ***city*** `{letterup}+(({punct}{space}|{space}){letterup}+)*{punct}?`
    - ***location*** `{letterup}{letterlow}*(({space}|{punct}|{punct}{space}){letterup}{letterlow}+)*{punct}?`
- `NOMET` deriva dalle regex *location* e *city* se e solo se la stringa letta ha dimensione 1.

Fase 3 - Analizzatore Sintattico

L'analizzatore sintattico è realizzato in Bison.
Viene definito un tipo di dato `union` come un intero `int int_t` e un puntatore a caratteri `char *chat_t`: nella tabella seguente vengono riportati i roken con tipi di attributo noto.

Token	Tipo di attributo
NOMEC	<string_t>
NOMEL	<string_t>
NOMET	<string_t>
NUM	<int_t>

L'assioma della grammatica divide l'analisi sintattica in 3 categorie che vengono analizzate da regole grammaticali differenti definite a partire dai token forniti dell'analizzatore lessicale e rappresentano le 3 sezioni dell'input:

- La prima parte dell'input viene gestita da una regola grammaticale ricorsiva destra che legge un numero variabile di Tappe. La gestione di ogni tappa è compito di una nuova serie di regole grammaticali che gestiscono singolarmente parti differenti: la città e la location, la data e i dettagli. La sezione dei dettagli è la più complessa poiché da essa viene presa la maggior parte delle variabili in input e poiché si permette all'utente di definire un numero variabile di tipi di biglietto e di categorie sconto.
- La seconda parte dell'input viene gestita da una regola grammaticale ricorsiva destra che legge un numero variabile di Prezzi. Ogni prezzo viene definito da una regola grammaticale a se stante che ne definisce la struttura.
- La terza parte dell'input viene gestita da una regola grammaticale ricorsiva destra che legge un numero variabile di Sconti

Fase 4 - Tabella dei Simboli

Per la conservazione delle tappe e dei loro dettagli si è preferito l'uso di una serie di liste concatenate poiché i valori delle tappe vengono letti dal programma insieme e si permette la definizione di un numero arbitrario dt tappe.

Per la conservazione delle informazioni degli sconti e dei prezzi si è preferito usare due differenti hashtable, `scontoHash` e `prezzoHash`, dato che non è detto che sconti e prezzi appaiano nello stesso ordine in cui vengono definiti nella prima parte dell'input. Vengono sviluppate delle funzioni accessorie per calcolare l'hash, inserimento e lookup

Funzioni definite per le azioni semantiche

- `setTappa($2, $5);`

  Creazione di una nuova tappa che ha come città il primo parametro e come location il secondo e inserimento nella lista concatenata adatta;

- `detBind();`

  Creazione di legame tramite puntatori tra l'ultima tappa letta (che non è ancora completa) e la lista di dettagli che si riferiscono a quella determinata tappa;

- `detInit($2);`

  Creazione di un nuovo dettaglio di tappa che ha come nome tipo di biglietto il singolo parametro e inserimento nella lista concatenata adatta;

- `catInit($1, $3);`

  Creazione di un nuovo dettaglio che ha come nome tipo di categoria il primo parametro e come numero di biglietti staccati il secondo parametro e inserimento nella lista concatenata adatta;

- `catBind();`

  Creazione di legame tramite puntatori tra l'ultimo dettaglio di tappa letto (che non è ancora completo) e la lista di dettagli che si riferiscono a quel determinato dettaglio della tappa;

- `setPrezzo($2, $4);`

  Viene richiamata quando è stato letto un prezzo e permette l'inserimento di questo valore nella hashtable corrispondente

- `setSconto($1, $3);`

  Viene richiamata quando è stato letto un prezzo e permette l'inserimento di questo valore nella hashtable corrispondente

Fase 5 - Esempi

File di input allegati in esempio

- **input.txt** (originale):
  Il codice genera un output corretto.

- **input2.txt**: categorie di sconto e tipi di biglietto in numero variabile.
  Il codice genera un output corretto mostrando che è capace di analizzare un numero variabile di `<nominativo categoria>: <numero di biglietti staccati>` e `<tipo biglietto>` purché la definizione sia consistente in numero e per identificatori con la terza e seconda parte dell'input, rispettivamente.

- **input3.txt**: formattazione di nome_città non rispettata.

  L'esecuzione del codice fallisce poichè il parser riconosce un input non corretto.

- **input4.txt**: formattazione di nome_location non rispettata.

  L'esecuzione del codice fallisce poichè il parser riconosce un input non corretto.