# RELAZIONE PROGETTO DI COMPILATORI

La consegna chiede di costruire un traduttore guidato dalla sintassi in grado di riconoscere un linguaggio che descrive i consumi d'acqua di alcuni utenti (contabilizzati in base a fasce di consumo).

Si deve tenere conto anche di:
1. costo dell'energia elettrica per l'autoclave
2. canone della fognatura
3. costo fisso di nolo del contatore

Obiettivo: calcolare il totale dovuto da ciascun utente comprendente anche l'IVA.

Il file in input è composto da 4 sezioni:
1. Viene specificato l'anno
2. Vengono specificate le fasce (non c'è un numero stabilito)
3. Elenco dei costi aggiuntivi
4. Lista di persone comprendendo: lettura al 1° gennaio e al 31 Dicembre.

Il programma deve riconoscere lessico e sintassi, e fornire in output per ogni utente indicato nell'input la spesa totale della bolletta.

**Input**
Anno 2020
?????
Fascia 1: 0,0 -> 46,0 -> € 0,3/mc
Fascia 2: 46,01 -> 58,0 -> € 0,8/mc
Fascia 3: 58,01 -> -> € 2,5/mc
*****
COSTI AGGIUNTIVI
costo dell'energia elettrica: € 0,1/mc
Canone fognatura: € 0,2/mc
Nolo contatore: € 1,5.
I.V.A.: 20%
#####
Maria Rosselli, RSSMRA77L07A546F, 116, 128
Francesco Amato, MTAFNC70H01H070L, 429, 459
Giuseppe Genovese, GNVGPP77S15G273H, 417, 473
Giorgia Conte, CNTGRG80E60G273I, 282, 293
Monica Ferrante, FRRMNC63R43C351T, 126, 157
Antonio Giusti, GSTNTN47L57I533E, 250, 285
Marta Scuderi, SCDMRT59D69G273L, 216, 297
Stefano Montalto, MNTSFN80M01A089A, 167, 255
Maria Grazia Leone, LNEMRG68L14G273C, 696, 732
Gian Luigi Caserta, CSRGLG77T23C708C, 531, 559
Alessandro La Barbera, LBRLSN81R11L131O, 157, 176

**Output** (lista di utenti composta da):

Nome Cognome -> spesa totale

Per prima cosa definiamo l'analizzatore lessicale con l'ausilio del linguaggio Flex, in cui utilizzo **%option noyywrap()** che comporta che non venga chiamata la funzione **yywrap()** dopo che è stato raggiunto un carattere di End-of-File e si assume che non esistano più file da analizzare.

# FLEX

## Sezione dichiarazioni

Nella sezione dichiarazioni aggiungerò il codice C delimitato da
%{ %}
in cui includo le librerie che mi interessano ed in particolare la libreria della tabella dei simboli, in particolare:

- stdlib.h
- string.h
- parser.tab.h ovvero uno degli output della compilazione del bison

In più aggiungo il prototipo della funzione di cui parlerò successivamente.

In più dichiaro alcune definizioni regolari per rendere più leggibile il file:

- **numero [0-9]** con {numero} si indicano tutti i numeri da 0 a 9.
- **lg [A-ZÀÁÈÉÌÒÙ]** per indicare tutte le lettere maiuscole e accentate.
- **lp [a-zàèéìòù]** per indicare tutte le lettere minuscole e accentate.

A partire dalle definizioni soprastanti creo le più complesse in cui definisco:

- **anno** composto da 4 numeri (è possibile inserire tra 1000 e 2999).
- **numero_intero** comprende i numeri interi
- **numero_reale** numeri reali
- **nominativo** comprende nome e cognome composti da lettera maiuscola lettere minuscole e spazio se ne hanno diversi.
- **codice_fiscale** composto da 6 lettere, 2 numeri, una lettera, 2 numeri, una lettera, 3 numeri e una lettera. Tutto maiuscolo.

## Sezione regole di traduzione

In questa sezione mi occupo delle regole di traduzione, in particolare di cosa fare dopo il match ad una regola.

Dichiaro tutte le stringhe "fisse" ovvero quelle che rimarranno in ogni file di input restituendole come token per il parser:

| Stringa | Token |
|---|---|
| "Anno" | ANNO |
| "?????" | SEP_SEZIONE |
| "Fascia" | FASCIA |

| | |
|---|---|
| “:” | DUEPUNTI |
| “,” | VIRGOLA |
| “->” | FRECCIA |
| “*****” | SEP_SEZIONE2 |
| “costo dell’energia elettrica:” | ELETTRICA |
| “Canone fognatura:” | FOGNATURA |
| “Nolo contatore:” | CONTATORE |
| “/mc” | METROCUBO |
| “I.V.A.:” | IVA |
| “%” | PERCENTUALE |
| “.” | PUNTO |
| “€” | EURO |
| “#####” | SEP_SEZIONE3 |

Mi occupo inoltre delle variabili di cui:

1. Per ogni **intero** assegno ad “yylval.intero” (la variabile corrispondente del parser per interi che definirò più avanti) la conversione in intero (funzione atoi()) della stringa yytext corrispondente dell’analizzatore lessicale.
2. Per ogni **reale** assegno ad “yylval.reale” la conversione in float (funzione atof()) della stringa yytext corrispondente, dopo aver fatto la sostituzione dei punti con le virgole chiamando la funzione sostituisci carattere() (in modo tale da poter convertire correttamente la stringa).
3. Per ogni **stringa** assegniamo ad “yylval.stringa” il puntatore al primo elemento della stringa yytext corrispondente.

| **Pattern** | **Token restituito** |
|---|---|
| {anno} | ANNO_VAL |
| {numero_intero} | INTERO |
| {numero_reale} | REALE |
| {nominativo} | NOMINATIVO |
| {codice_fiscale} | CODICEF |

E come ulteriori regole di traduzione inserisco lo spazio, la tabulazione l’invio a capo ed il punto in modo tale che se li trova o trova altro che non è congruo alle regole sopra lo scanner non compie nessuna azione.

## Sezione funzioni ausiliarie

In questa sezione ho inserito il corpo della funzione sostituisci_carattere(), in cui mi occupo di prendere 3 parametri: la stringa “yytext”, il carattere ‘,’ e il carattere ‘.’ in modo tale da restituire una stringa che al posto di avere la virgola ha il punto per poter effettuare correttamente la conversione.

# BISON

## Sezione dichiarazioni

Nella sezione dichiarazioni aggiungerò il codice C delimitato da
%{ %}
in cui includo le librerie che mi interessano ed in particolare la libreria della tabella dei simboli, in particolare:

- stdio.h
- stdlib.h
- string.h
- symb.h ovvero il file header della tabella dei simboli utilizzata successivamente per immagazzinare i dati che interessano.

In più ho aggiunto il prototipo della funzione **yylex** che corrisponde all’analizzatore sintattico, ed il prototipo **yyerror** una funzione che riporta gli errori riscontrati durante il parsing.

Utilizzo il costrutto **%union** per specificare l’intera collezione di tipi e permettere di scegliere di volta in volta il tipo più utile per i token.

In particolare nel costrutto dichiaro:
**int** intero;
**double** reale;
**char*** stringa;

e successivamente dichiaro i **token** che ho restituito dallo **scanner:**

**Token generici:**
ANNO SEP_SEZIONE FASCIA DUEPUNTI VIRGOLA FRECCIA SEP_SEZIONE2 ELETTRICA FOGNATURA CONTATORE METROCUBO IVA PERCENTUALE SEP_SEZIONE3 PUNTO EURO

**Token interi:**
ANNO_VAL INTERO

**Token reali**:
REALE

**Token stringhe**:
NOMINATIVO CODICEF

**Nota**: Riguardo i simboli non terminali si possono dichiarare con il costrutto **%type** ma nelle ultime versioni di bison vengono dichiarati implicitamente anche se non dichiarati.

In più dichiaro **%start** inizio, per definire l’assioma

# Sezione regole grammaticali

In questa sezione definisco le regole grammaticali che deve seguire il parser per riconoscere il linguaggio.
Seguendo la convenzione della programmazione in bison, ogni simbolo non terminale sarà scritto in minuscolo mentre ogni simbolo terminale sarà scritto in maiuscolo.

Utilizzando i token sovrastanti formo il linguaggio che deve seguire il parser per interpretare correttamente il file in input.

| | | |
|---|---|---|
| **start** | → | **inizio** |
| **inizio** | → | **anno_corrente** SEP_SEZIONE **descrizione_fasce** **fascia_speciale** SEP_SEZIONE2 **costi aggiuntivi** SEP_SEZIONE3 **lista_persone** |
| **anno_corrente** | → | ANNO ANNO_VAL |
| **descrizione_fasce** | → | **descrizione_fasce** **fasce** \| **fasce** |
| **fasce** | → | FASCIA **INTERO** DUEPUNTI **REALE** FRECCIA **REALE** FRECCIA EURO **REALE** METROCUBO |
| **fascia_speciale** | → | ε \| FASCIA **INTERO** DUEPUNTI **REALE** FRECCIA FRECCIA EURO **REALE** METROCUBO |
| **costi_aggiuntivi** | → | ELETTRICA EURO **REALE** METROCUBO FOGNATURA EURO **REALE** METROCUBO CONTATORE EURO **REALE** PUNTO IVA **INTERO** PERCENTUALE |
| **lista_persone** | → | **lista_persone** **NOMINATIVO** VIRGOLA **CODICEF** VIRGOLA **INTERO** VIRGOLA **INTERO** |
| **lista_persone** | → | **NOMINATIVO** VIRGOLA **CODICEF** VIRGOLA **INTERO** VIRGOLA **INTERO** |

L’assioma parte da **inizio** e segue le regole grammaticali definite
Le variabili che ci interessano e che sono da memorizzare nelle strutture dati sono marcate in grassetto.

In **fasce** richiamo la funzione **aggiungiFasciaInCoda($2, $4, $6, $9)** per aggiungere i valori della fascia, questa regola può essere richiamata più volte per aggiungere altre fasce.

In **fascia_speciale** richiamo la funzione **aggiungiFasciaInCodaS($2, $4, $8)** per aggiungere i valori della fascia tranne il limite superiore per indicare, come da consegna, il limite massimo della fascia. Questa regola può essere chiamata al più una volta perché possiamo avere un solo limite massimo come limite superiore altrimenti la fascia successiva a questa sarebbe sempre esclusa.

In **costi_aggiuntivi** richiamo la funzione **aggiungiCosti($3,$7,$11,$14)**, regola che deve essere chiamata esattamente una volta perché i costi aggiuntivi sono obbligatori.

In **lista_persone** richiamo la funzione **aggiungiPersona($2,$4,$6,$8)**, regola che può essere chiamata più volte e **aggiungiPersona($1,$3,$5,$7)** con parametri diversi se non c’è la chiamata ricorsiva “lista_persone”.

## Sezione funzioni ausiliarie

In questa sezione ho aggiunto la funzione principale **main()** avviata all’esecuzione del programma che controlla se il parsing del file in input ha dato successo (yyparse == 0), in caso affermativo:

- chiama la funzione di stampa dell’output richiesto
- chiama la funzione per liberare la memoria dalle strutture dati utilizzate dalla tabella dei simboli

altrimenti:

- viene chiamata in automatico la funzione **yyerror()** in cui si specifica il tipo di errore avvenuto.

## SYMBOL TABLE

Per creare la symbol table ho scritto 2 codici sorgente:

- l'header: **symb.h** che conterrà le dichiarazioni delle strutture dati utilizzate e tutti i prototipi delle funzioni
- il file C: **symb.c** in cui ci saranno i corpi delle funzioni utilizzate.

Ho utilizzato 3 strutture dati differenti per immagazzinare i dati:

1. **lista concatenata** per le fasce
2. **struttura normale** per i costi aggiuntivi
3. **hashtable** per la lista delle persone

La lista concatenata l'ho gestita con le funzioni:
**- aggiungiFasciaInCoda()** che si occupa di aggiungere la fascia nella coda della lista
**- aggiungiFasciaInCodaS()** richiama la funzione **aggiungiFasciaInCoda()** passando un valore predefinito (in questo caso 0.00) al valore **limiteSuperiore**, in modo tale da inserire come limite superiore il limite massimo ho utilizzato l'operatore ternario per far si che se si passa il parametro 0.00 viene sfruttata per correttezza la costante predefinita **DBL_MAX** della libreria **float.h**
**- trovaFascia()** serve per cercare la fascia corrispondente alla persona in questione, viene ripetuto un ciclo iterativo finché il conto inserito a parametro nella funzione è maggiore al limite superiore della fascia, appena esce dal ciclo restituisce la fascia attinente.

**-aggiungiCosti()** funzione utilizzata per aggiungere i costi aggiuntivi nella struttura normale

-**hash()** si occupa di calcolare il valore hash della stringa passata a parametro, insieme alla funzione **lookup()** si occupa di memorizzare i dati delle persone nell'hashtable.

**-aggiungiPersona()** serve ad aggiungere una nuova persona nella lista di persone; inizialmente **calcola l'hash** del codice fiscale per determinare la posizione dell'hashtable dove inserire la persona, se la persona non esiste già ne crea una nuova la inizializza con i dati ricevuti come parametri e la inserisce nella tabella hash della lista delle persone.

**-liberaMemoria()** funzione che si occupa di liberare la memoria occupata dall'hashtable e dalla lista concatenata di fasce.

**-stampaPersonePrezzi()** fa stampare appunto le persone affiancate ai costi per come richiede la consegna e richiama la funzione **calcoloCosto()** che per ogni persona calcola adeguatamente il prezzo da pagare.

*Alessandro Lo Bosco*