Relazione progetto

Giovanni Palmeri

1 Analisi lessicale

Come sappiamo l'analizzatore lessicale serve a riconoscere i vari lessemi del linguaggio e generare i token per ognuno di essi, a questo scopo viene incluso il file `parser.tab.h`, contenete i vari token definiti nel parser. Per quanto riguarda le espressioni regolari utilizzare per il riconoscimento dei lessemi abbiamo, partendo dai più semplici, i vari seperatori `?????`, `*****` e `#####` in aggiunta ai vari simboli utilizzati dell'input come: `€`, `%`, `.`, `,`, `:` e `->`, successivamente abbiamo quelle espressioni che ci permettono di identificare i vari costi aggiuntivi, ovvero : `costo dell'energia elettrica`, `Canone fognatura`, `Nolo contatore` e `I.V.A.`, infine abbiamo i pattern più complessi ovvero `digit [0-9]` che identifica un intero senza segno, `float {digit}+[,.]{digit}+` identifica appunto un float che può essere scritto con la virgola o con il punto, `name [A-Z][a-z]+([ ][A-Z][a-z]+)+` fa il match su cognome e nomi di un individuo ed infine `cf [A-Z]{6}[0-9]{2}[A-Z][0-9]{2}[A-Z][0-9]{3}[A-Z]` che rappresenta il codice fiscale dell'individuo.

2 Strutture dati

2.1 Strutture ausiliarie

Si tratta di strutture di supporto conteneti dati globali, ovvero che non cambiano da un individuo all'altro, si tratta delle strutture `Costi_aggiuntivi` e `Fascia`

Listing 1: Costi_aggiuntivi

```
typedef struct {
    float costo_elettricita;
    float canone_fognatura;
    float nolo_contatore;
    int iva;
} Costi_aggiuntivi;
```

Listing 2: Fascia

```
struct Fascia{
    int tier;
    float limite_inferiore;
    float limite_superiore;
    float costo_mc;
    struct Fascia *next;
};
```

le quali servono a contenere i dati sui vari costi aggiuntivi e sulle fasce specificate nel file di input, di seguito sono elencate le operazioni eseguibili su tali strutture dati:

Listing 3: Funzioni

```
void insertFascia(struct Fascia **fasce, int tier, float ls, float li,
    float costo);
void insertCanoneFognatura(Costi_aggiuntivi **costi_a, float cf);
void insertCostoEnergiaElettrica(Costi_aggiuntivi **costi_a, float ce);
void insertNoloContatore(Costi_aggiuntivi **costi_a, float nc);
void insertIVA(Costi_aggiuntivi **costi_a, float iva);
void printFasce(struct Fascia *fasce);
void printCosti(Costi_aggiuntivi *costi);
int checkRegularValue(struct Fascia *fasce, float li, float ls, int t);
```

2.2 Symbol Table

Per quanto riguarda la `SymbolTable` vera e propria, questa è definita nel seguente modo: `static struct SymbolEntry *hashTable[HASH_SIZE];` dove

Listing 4: Symbol

```
typedef struct Symbol {
    char *name;
    char *cf;
    int first_read;
    int last_read;
    float total;
} Symbol;
```

Listing 5: SymbolEntry

```
struct SymbolEntry {
    struct SymbolEntry *next;
    Symbol *info;
};
```

Si tratta quindi di una tabella hash la cui funzione di `hash` utilizza il metodo della divisione per il calcolo dell'indice dell'elemento, come possiamo vedere dal seguente codice

Listing 6: Hash

```
unsigned int hash(char *code) {
    unsigned hashval;
    for (hashval = 0; *code != '\0'; code++)
        hashval = *code + 31 * hashval;
    return hashval % HASH_SIZE;
}
```

Dato che l'informazione contenuta in questa tabella è basata su individui, come chiave è stato specificato il codice fiscale di questi ultimi. In definitiva le operazioni eseguibili su tale `SymbolTable` sono

Listing 7: Hash

```
unsigned int hash(char *code);
struct SymbolEntry *lookup(char *code);
int insert(char *name, char *cf, int fr, int lr);
void symbol_table_print();
```

```
void calculate_total(struct Fascia *fascie, Costi_aggiuntivi *ca);
void freeSymbol(Symbol *symb);
void freeSymbolEntry(struct SymbolEntry* entries);
void freeHashTable();
void print_total();
```

`lookup`, `insert` e `symbol_table_print` spiegano da sole il loro scopo, `calculate_total` calcola ed inserisce nel campo `total` della struttura `Symbol` il valore richiesto dalla consegna del progetto, passando a parametro la struttura contenente i vari costi aggiuntivi e la struttura che descrivere le fasce di prezzo, `print_total` stampa nel formato specificato dalla cosegna l'output richiesto, infine le varie funzioni `free` servono a liberare la memoria allocata per la hashTable.

3 Grammatica

Per quanto riguarda la progettazione della grammatica context-free, partiamo stabilendone l'alfabeto il quale sarà costituito dalle lettere maiuscolo e minuscolo dell'alfabeto italiano, dai numeri da 0 a 9 e
dai caratteri speciali `(?,*,#,-,>, ,, ., :, %, €, /)`. L'assioma è stato denominato `input` e la sua produzione è
`input->anno?????fasce*****costi_aggiuntivi#####lista_user`. Le altre produzioni sono riportate di seguito

Listing 8: Produzioni

```
anno -> ANNO INTEGER
fasce -> FASCIA INTEGER COLON INTEGER FRECCIA INTEGER FRECCIA EURO FLOAT
    MC fasce |

costi_aggiuntivi -> COSTO_ELETTRICO COLON EURO FLOAT MC CANONE_FOGNATURA
    COLON EURO FLOAT MC NOLO_CONTATORE COLON EURO FLOAT DOT IVA COLON
    INTEGER PERCENT DOT

lista_user -> NAME COMMA CF COMMA INTEGER COMMA INTEGER lista_user |
```

Si noti che nel parser effettivo sono state aggiunte delle produzioni in modo da poter specificare alcuni valori, nel file di input, in differenti formati, un esempio di ciò lo abbiamo con i limiti superiori ed inferiori le fasce che nel file di input possono presentarsi siamo come `unsigned int` che `float`.

4 Analisi sintattica

Come prima cosa abbiamo la dichiarazione dei token e di una struttura `union` che ci servirà per specificare il tipo di dato che un certo token rappresenterà, in particolare abbiamo

Listing 9: Union

```
%union {
    int ival;
    float fval;
    char *sval;
}
```

Listing 10: Type

```
%token <ival> INTEGER
%token <fval> FLOAT
%token <sval> CF NAME
```

Per quanto riguarda l'analisi sintattica vera e propria, in aggiunta a quanto detto nella sezione dove abbiamo parlato della grammatica, dobbiamo fare una precisazione sulle produzione `fasce` e `lista_user`. Queste infatti hanno come produzione anche ϵ, dovendo però assicurarci che queste due sezioni nel file input siano non nulle, si sono introdotte 2 ulteriori produzioni, rispettivamente `prima_fascia` e `primo_user`, le quali risolvono proprio questo problema. Inoltre possiamo soffermarci sui vari controlli che il parser esegue su determinate produzioni:

- Nella produzione di `fasce` vengono fatti 2 controlli, il primo si assicura, tramite la funzione `checkRegularValue` che il valore del limite superiore della fascia precedente sia uguale a quello del limite inferiore della fascia successiva, il secondo invece si assicura che il limite superiore di una fascia sia maggiore o uguale al suo limite inferiore.
- Nella produzione di `costi_aggiuntivi` viene fatto un controllo su valore dell'`IVA` che deve essere $0 < \textbf{IVA} \leq 100$.

Concludiamo dicendo che ovviamente nelle produzione di `costi_aggiuntivi`, `lista_user` e `fasce`, troveremo una chiamata alla corrispondente funzione di `insert` che andrà ad inserire i valori nelle strutture dati `struct Fascia *fasce = NULL;` e `Costi_aggiuntivi *ca = NULL;` inizializzate all'inizio del parser o nella tabella hash inizializzata in `symbletable.c`.

5 Note

Per l'esecuzione del progetto, su ambiente linux basta eseguire da terminale `./run.sh`, su windows bisogna aprire il terminale ed eseguire lo script `run.bat`.