Corso di Algoritmi e Strutture Dati
Laboratorio 4: Calcolo di espressioni aritmetiche

Si consideri un linguaggio di **espressioni aritmetiche su numeri interi con parentesi**. Informalmente, si tratta dell'insieme di tutte le stringhe (di caratteri, ovviamente) del tipo, per esempio:

```
12
( 3 + 4 )
( 7 * ( 4 + 24 ) )
( 66 + ( 56 - 5 ) )
( 88 * ( 9 * ( 3 * 17 ) ) )
```

Più formalmente (ma non troppo):

```
espr ::= numero | ( espr + espr ) | ( espr - espr ) | ( espr * espr )
```

(si tratta di una definizione ricorsiva; "`::=`" si legge "è definito come", mentre "`|`" vuol dire "oppure").

Per semplicità omettiamo l'operatore di divisione, perchè non è chiuso sugli interi. Inoltre omettiamo di considerare la proprietà associativa delle operazioni, per non dover considerare anche espressioni del tipo `( 43 + 5 + 8 )`. Dunque, nelle nostre espressioni, ogni operatore aritmetico prende sempre e solo due operandi e si fa abbondante uso di parentesi tonde.

Infine, per semplificare l'analisi dell'input, tra i vari elementi o *token* delle espressioni assumiamo la presenza di spazi, con funzione di separatori. Per esempio, `( 4 + 24 )` invece di `(4+24)`. In effetti tali spazi sono visibili anche nella definizione formale del linguaggio (vedi sopra). Ciascuna parentesi è un token, i numeri sono token (le singole cifre invece non lo sono), ciascun operatore aritmetico è un token, e i token sono tra loro separati da spazi. Notare che tutti i token sono stringhe di caratteri (sottostringhe dell'espressione) aventi ciascuno un proprio significato.

Un problema frequente è quello di **calcolare il valore di una espressione**, cioè convertire una espressione (ossia una stringa di caratteri appartenente al linguaggio definito come sopra) in un numero, assumendo che `+` rappresenti l'operazione di somma, `-` la sottrazione e `*` la moltiplicazione. Il calcolo del valore dovrebbe trasformare una stringa in un numero, ma solo se la stringa rispetta le regole della sintassi delle espressioni; altrimenti, il calcolo dovrebbe terminare indicando una condizione di errore.

Quando noi umani eseguiamo il calcolo a mano, partiamo dalle sottoespressioni più interne. Ogni volta che "risolviamo" una espressione interna, al suo posto sostituiamo il valore ottenuto e procediamo verso l'esterno. Ma si può procedere anche in un altro modo; se disponiamo di una struttura dati di tipo coda e di una struttura dati di tipo pila (uno *stack*, in gergo informatico) si può fare così:

1. **Fase 1: analisi lessicale**
   Estrarre uno dopo l'altro i token dalla stringa, inserendoli via via nella coda; ogni token viene etichettato con il suo tipo, che può essere: `PARENTESI_APERTA`, `PARENTESI_CHIUSA`, `NUMERO`, `OP_SOMMA`, `OP_SOTTRAZIONE`, `OP_MOLTIPLICAZIONE`.

Se si incontra un token che non ricade in alcuno dei tipi sopra elencati, si segnala errore lessicale ("simbolo sconosciuto") e l'algoritmo termina

2. **Fase 2: analisi sintattica e calcolo del valore**
   Estrarre uno dopo l'altro i token dalla coda, inserendoli via via sullo stack.
   Appena si incontra un token `PARENTESI_CHIUSA`, quello segnala la fine di una sottoespressione; allora tiriamo giù dallo stack gli ultimi cinque token inseriti.
   I token estratti dovrebbero essere esattamente, nell'ordine: un "`)`", un "numero", un operatore aritmetico, un altro "numero", e un "`(`"; se non è così, allora si segnala errore sintattico ("formula malformata") e l'algoritmo termina. Convertiamo i token di tipo `NUMERO` in numeri interi, eseguiamo l'operazione aritmetica opportuna, trasformiamo il risultato da numero a token (di tipo `NUMERO`) e inseriamo quest'ultimo sullo stack (in pratica abbiamo rimpiazzato una sottoespressione con il suo valore calcolato)

3. **Fase 3: controllo finale**
   quando la coda è vuota e l'ultimo token è stato elaborato, sullo stack dovrebbe essere rimasto un unico token di tipo `NUMERO`; quello rappresenta il risultato finale.

Per svolgere questa esercitazione occorre prima di tutto implementare un `Token`, cioè una `struct` contenente una stringa e una etichetta (quest'ultima implementata come `enum`). Fatto ciò, occorre implementare sia il tipo di dato `Coda` che il tipo di dato `Pila` di elementi di tipo `Token` (su questo non avete vincoli se non quelli di rispettare le operazioni presenti nel file header; potete riusare le implementazioni che avete fatto nei laboratori precedenti oppure usarne di altre).

Le implementazioni dei tipi di dato `Pila` e `Coda` devono poi essere usate per costruire due funzioni: una funzione `leggi()`, corrispondente alla fase 1 dell'algoritmo, e una funzione `calcola()`, corrispondente alle fasi 2 e 3.

La funzione `leggi()` avrà due parametri: uno di tipo IN, che rappresenta la stringa con l'espressione da valutare, e l'altro di tipo OUT, che rappresenta la coda contenente i token estratti. Essa inoltre restituisce un risultato booleano: `true` se tutti i token incontrati erano corretti, `false` altrimenti.

La funzione `calcola()` avrà due parametri: uno di tipo IN/OUT, che rappresenta la coda contenente i token (e che alla fine resterà vuota), e l'altro di tipo OUT, che rappresenta il risultato numerico. Essa inoltre restituisce un risultato booleano: `true` se l'espressione era sintatticamente corretta (dunque ha fornito un valore numerico), `false` altrimenti.

Per realizzare la funzione `leggi()` può essere utile implementare una funzione ausiliaria `prossimoToken()` che estragga da un `istringstream` il prossimo token (ossia la prima sottostringa consistente in caratteri non *whitespace* (spazi, a capo, tabulazioni, altri caratteri simili)).
Un modo per leggere il prossimo token "consumando" lo istringstream potrebbe essere:

```
bool prossimoToken(istringstream &iss, token &t)
{
    string s;
    iss >> s;
    if (!iss) {
        // non si possono leggere altri token, lo stream è terminato:
        // gestire la fine dell'input
    }
    ...
}
```

In generale gli `istringstream` sono stream di input con cui ci possiamo interfacciare in maniera analoga al familiare input da tastiera `std::cin`. Per una panoramica vedete, per esempio, `https://cplusplus.com/reference/sstream/istringstream/`.
Per realizzare la funzione `calcola()` può essere utile implementare due funzioni ausiliarie, per le trasformazioni da stringa a intero e da intero a stringa. Ci sono vari modi, ma il più pulito (come già visto ad IP) consiste nel passare dalla classe `string` alla classe `istringstream` del C++, e da questa al numero (il passaggio inverso avviene invece attraverso la classe `ostringstream`). Esempio:

```
#include <string>
#include <sstream>

using std::string;
using std::istringstream;
using std::ostringstream;

// da stringa a intero
string s = "123";
int n;
istringstream itmp(s);  // itmp: input stream con dentro la stringa "123"
                        // notare l'uso del costrutture con parametro ;-)
itmp >> n;              // ora n contiene il numero 123

// da intero a stringa
n = 456;
ostringstream otmp;
otmp << n;              // otmp: output stream con dentro il numero 456
s = otmp.str();         // da output stream a stringa e' immediato
                        // adesso s contiene la stringa "456"
```

*

Le funzioni `leggi()` e `calcola()` devono poi essere utilizzate nel `main()`. Il `main()` deve acquisire da standard input la stringa con l'intera espressione da valutare, e poi visualizzare il risultato numerico, oppure indicare che è presente un errore lessicale o sintattico nell'espressione.