ASD 23/24 - Laboratorio 4

In questo laboratorio viene richiesto di implementare una funzione che calcola il valore di *espressioni aritmetiche su numeri interi con parentesi*.

1 Spiegazioni generale

Si consideri un linguaggio di *espressioni aritmetiche su numeri interi con parentesi*. Informalmente, si tratta dell'insieme di tutte le stringhe (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 pila (uno *stack*, in gergo informatico) si può fare così:

1. **Fase 1: analisi lessicale**
   Si usa una funzione che estrae una dopo l'altro i token dalla stringa, ogni token viene etichettato con il suo tipo, che può essere: `PARENTESI_APERTA`, `PARENTESI_CHIUSA`, `NUMERO`, `OP_SOMMA`, `OP_SOTTRAZIONE`, `OP_MOLTIPLICAZIONE`.
   Per il caso dove il token è di tipo `NUMERO`, registriamo il valore in un intero transformando la stringa associata. Se si incontra un token che non ricade in alcuno dei tipi sopra elencati, si segnala errore lessicale e l'algoritmo termina

2. **Fase 2: analisi sintattica e calcolo del valore**
   Ogni token estratto dalla stringa, viene inserito via via sullo stack.
   Appena si incontra un token `PARENTESI_CHIUSA`, quello segnala la fine di una sottoespressione; allora estraiamo 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 e l'algoritmo termina. Quindi 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. Se non è cosi si segnala un errore sintattico.

2 Analisi lessicale

2.1 Materiale dato

Nel file `asd-lab4-traccia.zip`, trovate:

- Un file `ASD-token.h` contenente le definizioni di tipo dato per i token e l'intestazione della funzione che estra i token di una stringa
- Un file `ASD-token.cpp` dove dovete scrivere l'implementazione della funzione richiesta

2.2 Funzione da implementare

Il file `ASD-token.h` contiene la definizione della struttura `token` e il prototipo della funzione che dando una `string` produce il prossimo token.

```
//Tipi di dato semplici
enum kind {PARENTESI_APERTA, PARENTESI_CHIUSA, NUMERO, OP_SOMMA, OP_SOTTRAZIONE,
           OP_MOLTIPLICAZIONE};

struct token {
  int val;
  kind k;
};

/************************************************/
/*       prototipi di funzioni da implementare    */
/************************************************/
//funzione che estrae il prossimo token della string st
//lo mette in tok e modifica st
//ritorna true se c'era un token da estrarre,
//ritorna false se non c'era da estrarre e si e arrivato alla fine di st,
//solleva una eccezione di tipo string se legge un token di tipo sconosciuto
bool nextToken(std::string &st, token &tok);
```

In pratica, la funzione `bool nextToken(std::string &st, token &tok)` cerca il prossimo token in `st` cancellando gli spazi, se lo trova lo mette in `tok` con il tipo corrispondente nel campo `k` e il valore nel campo `val` se il token è un numero, e in fine rimuove i caratteri letti da `st`. Se la funzione arriva alla fine di `st` senza trovare un token, i.e. se `st` è vuota o contiene solo degli spazi, la funzione ritorna `false`, se trova un token ritorna `true` e se invece incontra una sequenza di caratteri che non corrisponde ad un token solleva un eccezione.
Ad esempio:

- se abbiamo `st="  (   11  "`, allora dopo la chiamata a `nextToken(st, tok)` avremo `PARENTESI_APERTA` in `tok.k` e il valore di `st` sarà `"   11  "` e la chiamata ritornerà `true`;
- se abbiamo `st="   11  "`, allora dopo la chiamata a `nextToken(st, tok)` avremo `NUMERO` in `tok.k`, 11 in `tok.val` e il valore di `st` sarà `"  "` e la chiamata ritornerà `true`;
- se abbiamo `st="  "`, allora la chiamata a `nextToken(st, tok)` ritornerà `false`;
- se abbiamo `st="(1 "` o `st="( AB "` , allora la chiamata a `nextToken(st, tok)` solleverà un eccezione di tipo `string`.

Per programmare questa funzione, potete usare le funzione della libreria `string` di C++ come

- `erase` (cf. https://cplusplus.com/reference/string/string/erase/
- `substr` (cf. https://cplusplus.com/reference/string/string/substr/)
- `stoi` (https://cplusplus.com/reference/string/stoi/): attenzione che prima di richiamare `stoi` va verificato che la stringa sia composta solo da numeri; oppure potete usarla insieme alla successiva `to_string` per fare un check che la conversione da stringa a numero sia corretta (vedi test finali con "numeri" quali 9.4 o 40.ABC)
- `to_string` (https://cplusplus.com/reference/string/to_string/)

3 Pila di token

3.1 Materiale dato

Nel file `asd-lab4-traccia.zip`, trovate:

- Un file `ASD-stack.h` contenente le definizioni di tipo dato per la pila di token e l'intestazione delle funzione
- Un file `ASD-stack.cpp` dove dovete scrivere l'implementazione delle funzione richieste

3.2 Da implementare

Come lo potete constatare, nel file `ASD-stack.h`, il tipo `Stack`, che sarà usato per fare una pila di token, non è corretto.

```
// Implementa STACK
namespace stack{
    // tipo base
    typedef token Elem;

    typedef int Stack; //DA CAMBIARE

    /***********************************************/
    /*       prototipi di funzioni da implementare    */
    /***********************************************/

    /* restituisce lo stack vuoto */
    Stack createEmpty();

    /* restituisce true se lo stack e' vuoto */
    bool isEmpty(const Stack&);

    /* aggiunge elem in cima (operazione safe, si puo' sempre fare) */
    /* NB: se stack implementato con array dinamico,
       quando necessario implementare ridimensionamento in espansione*/
    void push(const Elem, Stack&);

    /* toglie dallo stack l'ultimo elemento e lo restituisce */
    /* se lo stack e' vuoto solleva una eccezione di tipo string */
    /* NB: se stack implementato con array dinamico,
       quando necessario implementare ridimensionamento in contrazione*/
    Elem pop(Stack&);
}
```

Aspettiamo che date una definizione giusta per il tipo `Stack` e che implementate nel file `ASD-stack.cpp` le funzione usando questo tipo.

4 Analisa sintattica

4.1 Materiale dato

Nel file `asd-lab4-traccia.zip`, trovate:

- Un file `ASD-aritexpr.h` contenente l'intestazione della funzione che calcola il valore di una espressione aritmetica
- Un file `ASD-aritexpr.cpp` dove dovete scrivere l'implementazione di questa funzione.

4.2 Funzione da implementare

Per questa parte, dovete programmare la funzione `int compute_arithmetic_expr(const std::string& st)` che calcola il valore del espressione aritmetica contenuta in `st`. Se `st` non contiene un espressione aritmetica (i.e. non è nella forma corrispondente ad un espressione aritmetica) allora la funzione solleverà un espressione di tipo `string` con il valore `"Lexical Error"` o `"Syntactical Error"` secondo se l'algoritmo incontra un errore lessicale o sintattico.

5 Tests automatici

Nel file `ASD--aritexpr-test.cpp`, abbiamo programmato una sequenza di tests che si eseguono automaticamente e dove verifichiamo che la funzione `compute_arithmetic_expr` si comportano bene. Per usare questo programma, potete compilarlo cosi: `g++ -Wall -std=c++14 ASD-token.cpp ASD-stack.cpp ASD-aritexpr.cpp ASD-aritexpr-test.cpp -o ASD-aritexpr-test` e poi eseguirlo con `./ASD-aritexpr-test`.

6 Consegna

Per la consegna, creare uno `zip` con tutti i file forniti; in particolare con il file `ASD-token.cpp`, `ASD-stack.h`, `ASD-stack.cpp` e `ASD-aritexpr.cpp` da voi modificati.