LABORATORIO 4
CALCOLO DI ESPRESSIONI ARITMETICHE

Algoritmi e strutture dati

CONSIDERAZIONI

- **Molto più complesso** rispetto ai laboratori precedenti
  - Richiede un livello di autonomia maggiore
  - Richiede più tempo; è verosimile che per completarlo sia necessario sacrificare parte del tempo libero nel fine settimana: all'università può capitare spesso...

- **Laboratorio 'diverso'**
  - Si vede un **possibile utilizzo** di due tipi di dato che abbiamo studiato:
    - **Pile** e **Code** -- possibilmente implementate con la struttura dati più efficiente tra quelle che abbiamo analizzato
  - Ci permette di parlare di **sintassi** e **semantica** delle espressioni aritmetiche nonché di **analisi lessicale** ed **analisi sintattica**: due attività che g++ esegue ogni volta che compilate!

ESPRESSIONI ARITMETICHE

- Si consideri il linguaggio delle espressioni aritmetiche formate da
  - Numeri interi positivi a più cifre
  - Operatori +, -, *
  - Parentesi aperte e chiuse

- Esempi:
  - 12
  - (3 + 4)
  - (7 * (4 + 24))
  - ((56 - 5) + 2)
  - (88 * (9 * (3 * 17)))

SEMPLIFICAZIONI

Token
Numeri interi positivi (es. 23)
Operatori +, -, *
Parentesi (,)

- No operatore di divisione
  - (10 / 2) ---> ERRORE!
- No proprietà associativa
  - (3 + 5 + 7) ---> ERRORE!
- Tra un **token** e l'altro ci deve essere uno spazio
  - (3+5) ---> ERRORE!
- Token = costrutto valido/riconosciuto del linguaggio

DEFINIZIONE DI ESPR

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

- Questa è una definizione ricorsiva: si chiama BNF ("Backus–Naur form" o "Backus normal form") e definisce la **sintassi** di un'espressione aritmetica semplificata descritta prima
  - '::=' si legge **è definito come**
  - '|' vuol dire **oppure**

(INCISO)

- Anche la sintassi dei linguaggi di programmazione può essere espressa mediante una BNF
- https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm

<selection-statement> ::=

if (<expression>) <statement>

| if (<expression>) <statement> else <statement>

| switch (<expression>) <statement>

<iteration-statement> ::= while (<expression>) <statement>

| do <statement> while (<expression>) ;

| for ({<expression>}? ; {<expression>}? ;

{<expression>}?) <statement>

PROBLEMA DA RISOLVERE

- Calcolare il valore (la **semantica**!) di un'espressione:
  - **es:** ((10 + 1) * 2) ---> risultato = 22
  - Cioè associare ad una espressione sintatticamente corretta il numero naturale che tale espressione rappresenta, assumendo che il simbolo + si interpreti nell'operazione di somma tra naturali, - nella sottrazione e * nella moltiplicazione
- Come lo farebbe un umano:

((10 + 1) * 2) ----> (11 * 2) ----> 22

Si risolve

Si parte dalle sottoespressioni piu interne

APPLICAZIONE SOFTWARE PER CALCOLO DI ESPRESSIONI

Dovete implementare un'applicazione software che

- Presa in input una stringa:
  - Se la stringa è un'**espressione sintatticamente corretta** calcola e stampa a video il valore dell'espressione
    - ((10 + 1) * 2) risultato = 22
  - Se la stringa **non è** un'espressione sintatticamente corretta stampa a video (sul canale "standard error") 'errore sintattico'
    - (10 + 1 risultato = errore sintattico
  - Se la stringa contiene **costrutti non presenti** nel linguaggio delle espressioni stampa 'errore lessicale'
    - (10 % 2) risultato = errore lessicale

IDEA DELL'ALGORITMO

- Algoritmo a <u>tre fasi</u> che utilizza una **Coda** e un **Pila**

- **Fase 1: analisi lessicale**
  - Si estraggono uno dopo l'altro i token dalla stringa, inserendoli via via nella **Coda**; ogni token viene etichettato con il suo tipo (es. NUMERO). Se si incontra un token non riconosciuto si segnala errore lessicale
- **Fase 2: analisi sintattica e calcolo**

Si estraggono uno dopo l'altro i token dalla **Coda**, inserendoli via via sulla **Pila**. Quando sulla **Pila** viene messa una parentesi chiusa ')' allora si estraggono dalla stessa **Pila** 5 token, si effettua il calcolo della sotto-espressione (SE POSSIBILE!) e si inserisce il risultato sulla **Pila** (sempre come token)

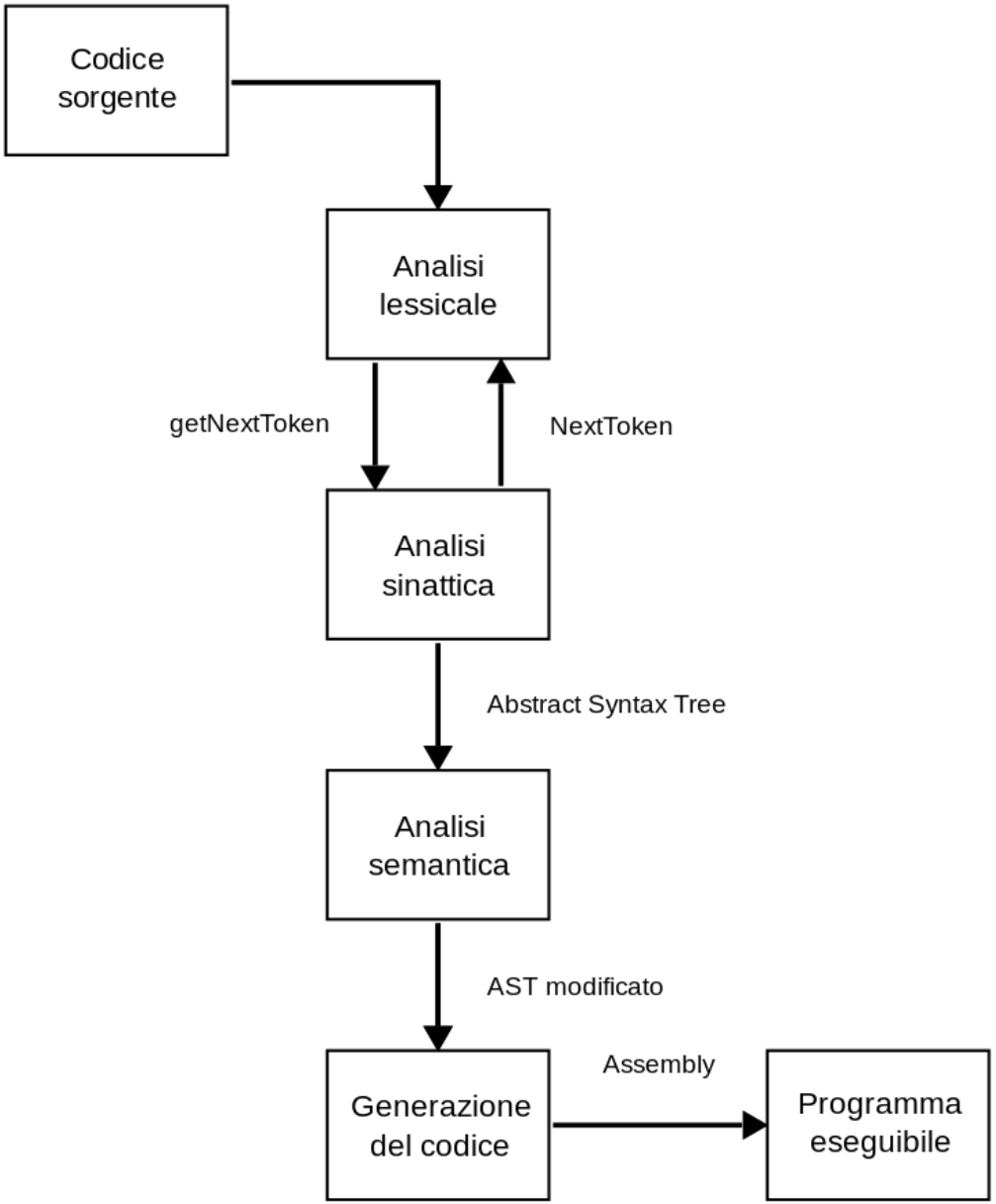
- **Fase 3: risultato**
  - Quando la **Coda** è vuota e l'ultimo token e stato elaborato, se la stringa era sintatticamente corretta allora sulla **Pila** dovremmo avere il risultato finale

(INCISO)

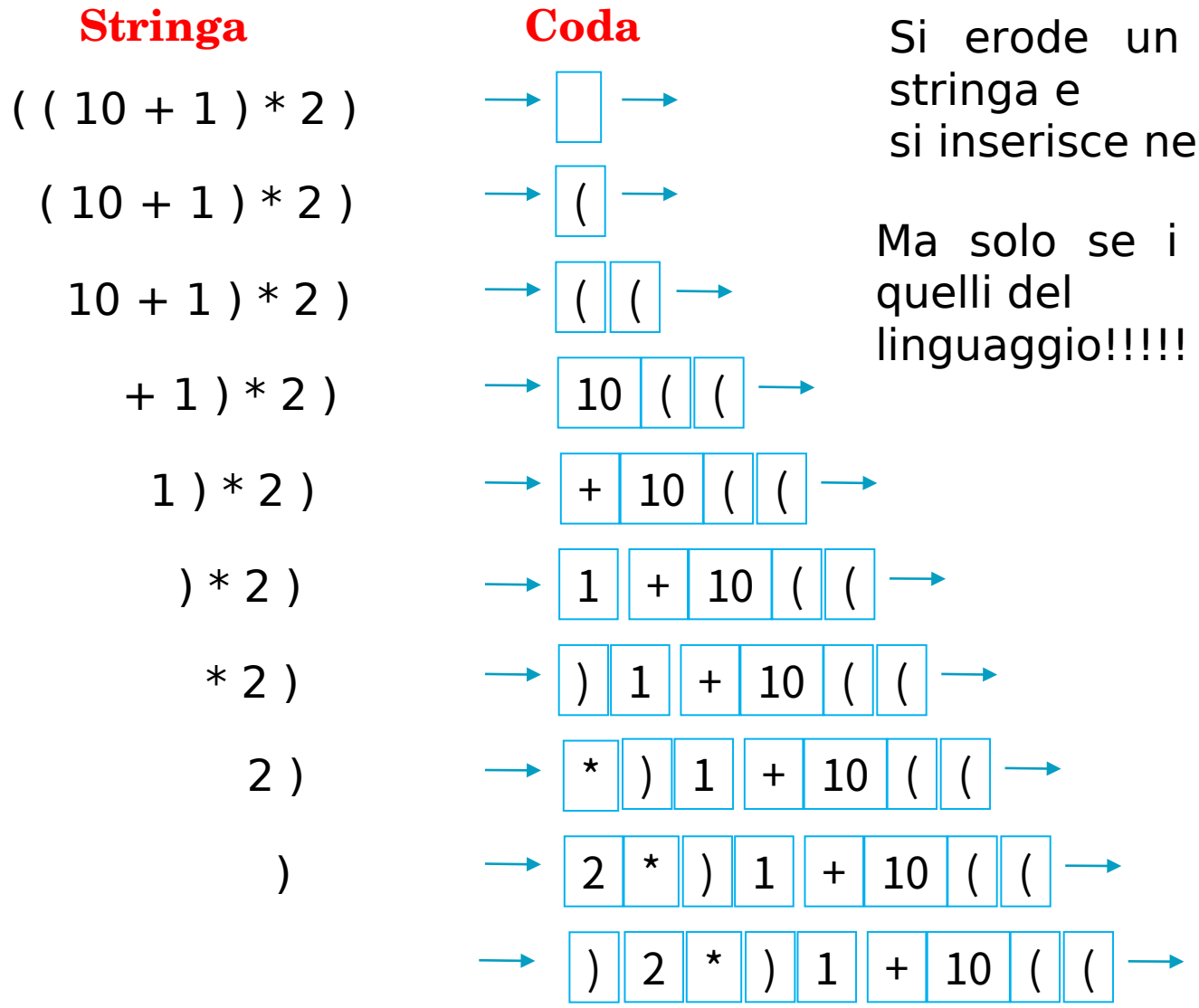
- Anche i compilatori dei linguaggi di programmazione operano seguendo i passi previsti in questo laboratorio
- **Analisi lessicale.** Attraverso un analizzatore lessicale, spesso chiamato scanner o lexer, il compilatore divide il codice sorgente in tanti pezzetti chiamati token. I token sono gli elementi minimi (non ulteriormente divisibili) di un linguaggio, ad esempio parole chiave (for, while), nomi di variabili (pippo), operatori (+, -, «).
- **Analisi sintattica.** L'analisi sintattica prende in ingresso la sequenza di token generata nella fase precedente ed esegue il controllo sintattico, effettuato attraverso una grammatica.
- **Analisi semantica.** L'analisi semantica si occupa di controllare il significato delle istruzioni presenti nel codice in ingresso. Controlli tipici di questa fase sono il type checking, ovvero il controllo di tipo, controllare che gli identificatori siano stati dichiarati prima di essere usati e così via. Come supporto a questa fase viene creata una tabella dei simboli (symbol table) che contiene informazioni su tutti gli elementi simbolici incontrati quali nome, scope, tipo (se presente) etc.

(da https://it.wikipedia.org/wiki/Compilatore)

(INCISO)


Codice sorgente
Analisi lessicale
getNextToken
NextToken
Analisi sinattica
Abstract Syntax Tree
Analisi semantica
AST modificato
Assembly
Generazione del codice
Programma eseguibile

ESEMPIO (ANALISI LESSICALE)



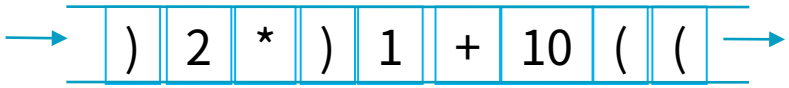
Si erode un token dalla stringa e
si inserisce nella coda ...

Ma solo se i token sono quelli del linguaggio!!!!!

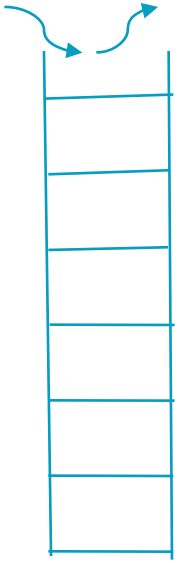
L'analisi lessicale si arresta se viene trovato un token sconosciuto

ESEMPIO (ANALISI SINTATTICA + CALCOLO)

Coda

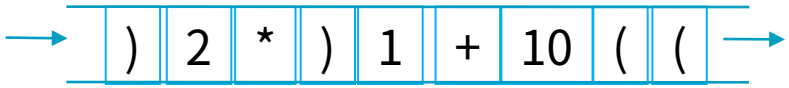


Pila

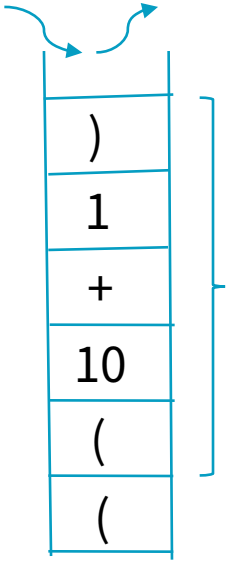


ESEMPIO (ANALISI SINTATTICA + CALCOLO)

Coda



Pila

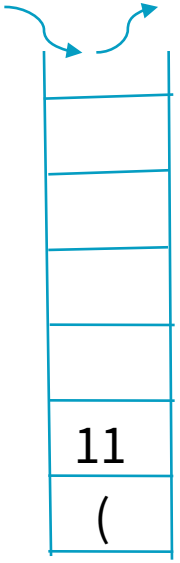


Provo a sostituire gli ultimi 5 token con il valore

ESEMPIO (ANALISI SINTATTICA + CALCOLO)

Coda

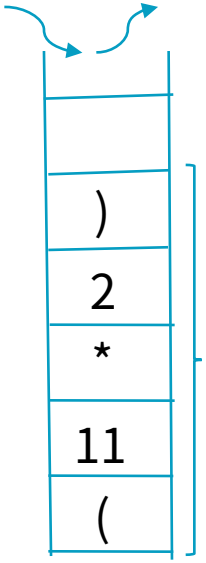
Pila



ESEMPIO (ANALISI SINTATTICA + CALCOLO)

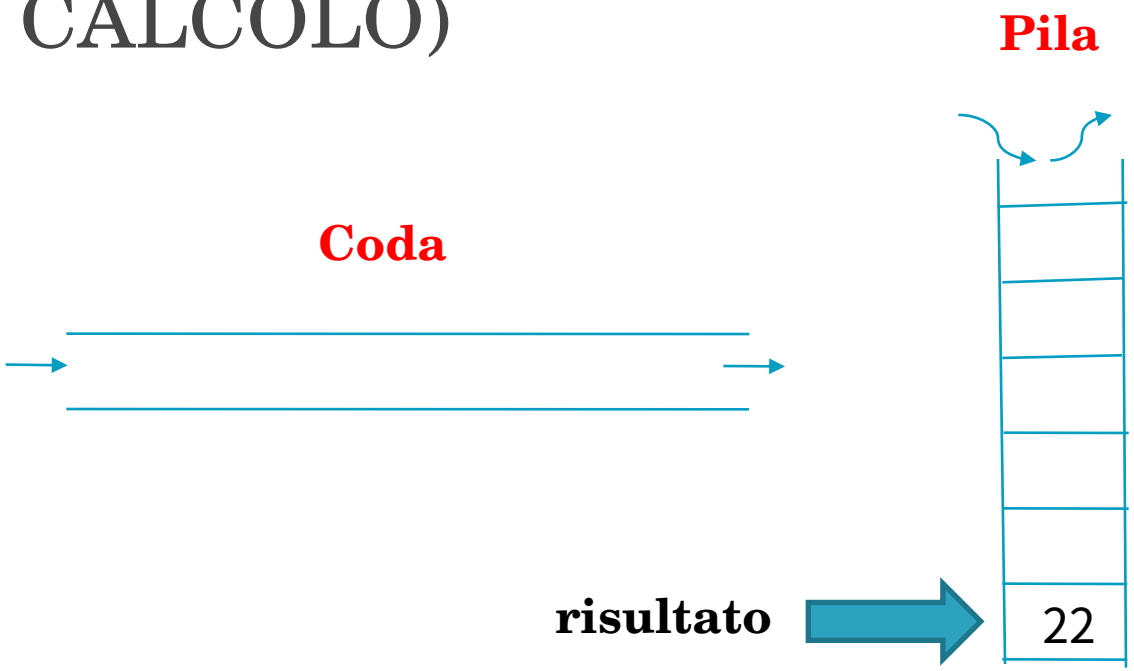
Coda

Pila



Provo a sostituire gli ultimi 5 token con il valore

ESEMPIO (ANALISI SINTATTICA + CALCOLO)

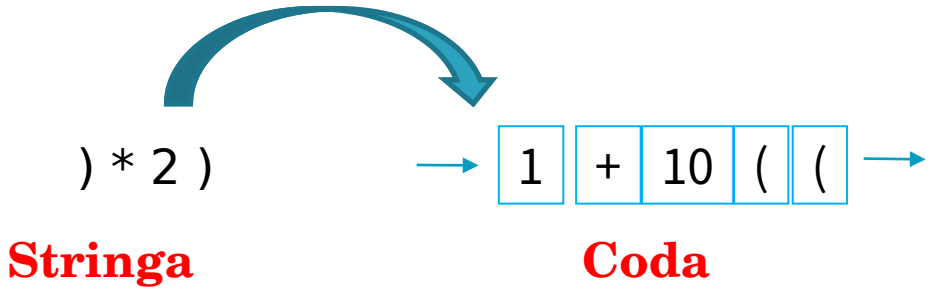


((10 + 1) * 2)

OPERATIVAMENTE

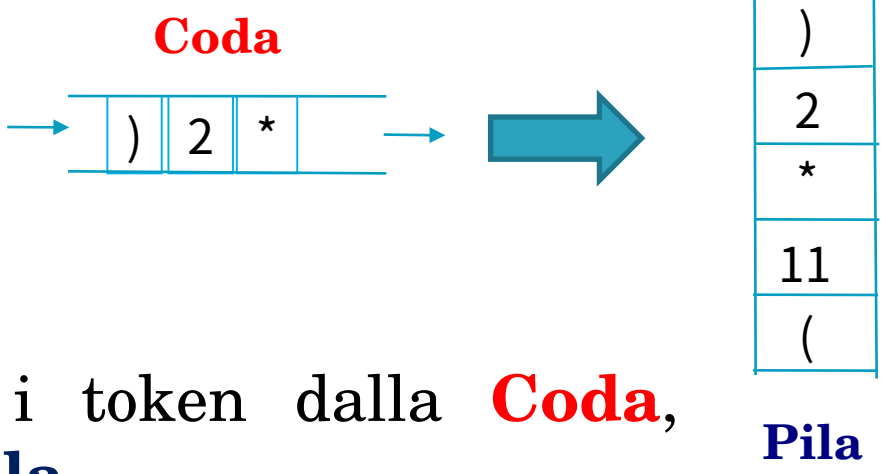
- **token.h**, **token.cpp**
  - Header e implementazione del tipo token
- **queue.h**, **queue.cpp**
  - Header e implementazione del tipo di dato Coda di token
- **stack.h**, **stack.cpp**
  - Header e implementazione del tipo di dato Pila di token
- **main.cpp**
  - Definisce le due funzioni **leggi()** e **calcola()** e il main che esegue la lettura e la valutazione dell'espressione

LEGGI()



- Si occupa della fase 1
- Estrae uno dopo l'altro i token dalla stringa, inserendoli via via nella **Coda**
  - ogni token viene etichettato con il suo tipo
    - es. NUMERO
- Restituisce un risultato booleano: **true** se tutti i token incontrati sono corretti, **false** altrimenti
- Per realizzare la funzione leggi() consigliamo di implementare una funzione ausiliaria **prossimoToken()**
  - In grado di estrarre il prossimo token (ossia la sotto-stringa fino al prossimo spazio)

CALCOLA()



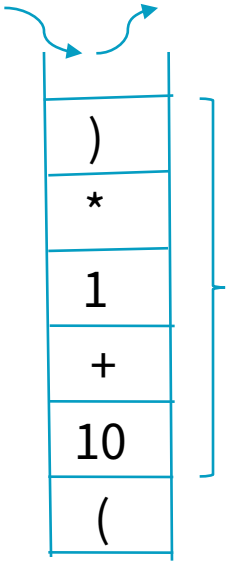
- Si occupa della fase 2
- Estrae uno dopo l'altro i token dalla **Coda**, inserendoli via via sulla **Pila**
  - Appena si incontra un token ‘)’, quello segnala la fine di una sottoespressione; allora si prelevano dalla pila gli ultimi cinque token inseriti e si calcola il valore
- Restituisce un risultato booleano: true se l'espressione era sintatticamente corretta (dunque ha fornito un valore numerico), false altrimenti

ERRORE SINTATTICO

(10 + 1 *)

Coda

Ok analisi lessicale!



Provo a sostituire gli ultimi 5 token con il valore

ERRORE
non è sotto-espressione corretta

OPERAZIONI SUI CARATTERI

- Come si fa a verificare se un carattere è una cifra?
- **c.isdigit()**

TOKEN

- Dovremo convertire da stringa a numero intero per fare i calcoli e viceversa per mettere in cima allo stack il token che contiene il risultato del calcolo: il token è definito come

```
struct token {
  string val;
  kind k;
};
```

Il campo val del token è una stringa

STRINGHE E STRING STREAM

- Possiamo convertire le stringhe in **string stream** di input e output, e lavorare su di loro come facciamo con **std::cin** (input stream), **std::cout** e **std::cerr** (output stream) dopo aver incluso la libreria **sstream**
  - **std::cerr** rappresenta lo **standard error**, canale riservato ai messaggi di errore (di default, stampato su video come **cout**)
- Dalla stringa **s** all'input stream **iss**:

  std::istringstream iss(s);
- Dalla stringa **s** all'output stream **oss**:

  std::ostringstream oss(s);

OPERAZIONI SULLE STRINGHE

- **Da stringa a numero intero**: sia **s** una stringa che contiene solo caratteri numerici ed **n** un int; usando il namespace std::istringstream possiamo scrivere

```
istringstream iss(s);
iss >> n;
```

- L'effetto sarà di inserire nella variabile **n** il numero che corrisponde alle cifre in **s** interpretate come un numero decimale

OPERAZIONI SULLE STRINGHE

- **Da numero intero a stringa**: sia n un numero intero positivo ed **s** una stringa; usando il namespace std::ostringstream possiamo scrivere

```
ostringstream otmp;
otmp << n;
s = otmp.str();
```

- L'effetto sarà di inserire nella stringa **s** i caratteri numerici che rappresentano il numero **n** in formato decimale

STRINGHE E STRING STREAM

- Le stringhe sono molto simili a **vector di char**: posso accedere al carattere i-esimo della stringa **s** usando la sintassi **s[i]** (s[0] è il primo carattere della stringa, s[1] il secondo, etc).
- Il primo parametro di **prossimoToken()** è un **istringstream**, che legge uno a uno da stringa gli input esattamente come fa **std::cin**; se **iss** è un istringstream quindi:
  - **iss >> s**, se **s** è una stringa, inserisce nella variabile s il prossimo gruppo di caratteri che non sono "whitespace" (spazi, a capo, tabulazioni, e altri "rari")
  - Possiamo verificare se c'è stato un errore (se assegnamo a una stringa, solo perché l'input è finito) con **if (!iss) {***(gestione della fine input)***}**

STRINGHE E STRING STREAM

https://cplusplus.com/reference/string/string/
https://cplusplus.com/reference/sstream/istringstream/
https://cplusplus.com/reference/sstream/ostringstream/

COME ORGANIZZARE IL LAVORO

Per prima cosa progettate ed implementate le operazioni sui token. Non serve che abbiate implementato tutto il resto del programma per verificare se riuscite a leggere un token da una stringa: potete modificare (temporaneamente) il main per chiamare direttamente le funzioni sui token e verificarne il comportamento. Avrete bisogno di implementare funzioni ausiliarie non richieste (ad es, per stampare un token, altrimenti non potrete vedere se viene correttamente preso dalla stringa in input).

Non passate ad implementare nient'altro finché le operazioni sui token non sono complete, funzionanti, adeguatamente testate.

COME ORGANIZZARE IL LAVORO

Poi implementate le operazioni dei TDD Pila e Coda di Token scegliendo strutture dati “furbe”. Anche in questo caso non serve che abbiate implementato le funzioni leggi() e calcola() per fare un testing adeguato: chiamate da main le funzioni su Pila e Coda stampandone il risultato ed effettuate un testing esaustivo.

Non passate ad implementare nient'altro finché le operazioni su Pila e Coda di Token non sono complete, funzionanti, adeguatamente testate.

COME ORGANIZZARE IL LAVORO

Solo quando le operazioni su Token, Pila e Coda sono state implementate e testate, potete progettare ed implementare l'algoritmo di lettura (analisi lessicale) e calcolo (analisi sintattica e valutazione) e farne un testing adeguato.