3. Analisi lessicale

- Σ: rappresenta un **alfabeto**, ovvero un insieme finito di simboli. Ad esempio, potrebbe essere $\Sigma = \{a, b\}$
- Σ^*: denota l'insieme di tutte le possibili stringhe finite che si possono formare usando i simboli dell'alfabeto Σ. Include tutte le combinazioni di questi simboli, inclusa la stringa vuota
- $\epsilon \in \Sigma^*$: è la **stringa vuota**, una stringa di lunghezza zero che non contiene simboli
- $L \subseteq \Sigma^*$: rappresenta un **linguaggio**, cioè un sottoinsieme delle stringhe appartenenti a Σ^*. Può includere alcune o tutte le stringhe che si possono formare dall'alfabeto Σ

Alcuni esempi di linguaggi:

- $\emptyset$: è l'insieme vuoto, ovvero il linguaggio che non contiene nessuna stringa.
- $\{\epsilon\}$: è un linguaggio che contiene solo la stringa vuota.
- Σ: rappresenta un linguaggio composto da tutte le stringhe di lunghezza 1, corrispondenti ai singoli simboli dell'alfabeto (con un piccolo abuso di notazione).
- Σ^*: è un linguaggio che contiene tutte le stringhe di qualsiasi lunghezza (comprese la stringa vuota) formate con i simboli dell'alfabeto Σ. Se $\Sigma \neq \emptyset$, questo linguaggio è infinito.

Il **problema di decisione** per un linguaggio L (noto anche come problema di riconoscimento o appartenenza) consiste nel trovare un algoritmo che, data una qualunque stringa $s \in \Sigma^*$, determini (in tempo finito) se la stringa s appartiene a L, ovvero se $s \in L$.

L’insieme RExpr delle RE (espressioni regolari)

Un'espressione regolare (RExpr) è costruita secondo le seguenti regole:

Simboli di base

- $\epsilon \in \mathrm{RExpr}$: La stringa vuota è un'espressione regolare.
- $\forall a \in \Sigma, a \in \mathrm{RExpr}$: Ogni simbolo dell'alfabeto Σ è un'espressione regolare.

Costruzioni derivate

Se $e \in \mathrm{RExpr}$, allora:

- $(e) \in \mathrm{RExpr}$: L'uso delle parentesi è consentito per raggruppare le espressioni.

- $e^* \in \mathrm{RExpr}$: La chiusura di Kleene dell'espressione e è un'espressione regolare.

Se $e_1 \in \mathrm{RExpr}$ e $e_2 \in \mathrm{RExpr}$, allora:

- $e_1 e_2 \in \mathrm{RExpr}$: La concatenazione di e_1 ed e_2 è un'espressione regolare.
- $e_1 | e_2 \in \mathrm{RExpr}$: L'alternanza (o unione) tra e_1 ed e_2 è un'espressione regolare.

Nota: La chiusura di Kleene (*) ha la precedenza sulla concatenazione, mentre la concatenazione ha la precedenza sull'alternanza ($|$).

Un linguaggio L definito da un'espressione regolare, $L : \mathrm{RExpr} \to \mathcal{P}(\Sigma^*)$, viene costruito seguendo alcune regole per generare sottoinsiemi dell'insieme di tutte le stringhe possibili (Σ^*). Ecco le regole fondamentali:

- **Stringa vuota**: $L(\epsilon) = \{\epsilon\}$
- **Simbolo singolo**: $L(a) = \{a\}$
- **Concatenazione**: $L(e_1 e_2) = L(e_1)L(e_2) = \{s_1 s_2 \mid s_1 \in L(e_1), s_2 \in L(e_2)\}$
- **Unione**: $L(e_1 \mid e_2) = L(e_1) \cup L(e_2)$
- **Chiusura di Kleene**: $L(e^*) = \bigcup_{i=0}^{\infty} L(e)^i$
- **Raggruppamento**: $L((e)) = L(e)$

Estensioni aggiuntive

- **Chiusura positiva**: $L(e^+) = \bigcup_{i=1}^{\infty} L(e)^i$
- **Opzionalità**: $L(e?) = \{\epsilon\} \cup L(e)$
- **Classe di simboli / range**: $L([a_1 \ldots a_n])$ e $L([a_1 - a_n])$ rappresentano rispettivamente l'insieme di simboli specifici $a_1, \ldots, a_n$ o l'intervallo di simboli tra a_1 e a_n.
- **Complemento di classe/range**: $L([\hat{}a_1 \ldots a_n])$ e $L([\hat{}a_1 - a_n])$ rappresentano tutti i simboli che non sono inclusi nell'insieme specificato.

Token

I **token** nei linguaggi di programmazione rappresentano gli elementi di base che costituiscono il codice sorgente. Essi possono essere classificati nelle seguenti categorie:

Parole chiave. Termini riservati del linguaggio, come `if`, `then`, `else`, `while`, ecc. Nelle espressioni regolari, queste possono essere rappresentate tramite l'alternanza: `if | then | else | while | ...`. In teoria, vengono raccolte sotto una categoria sintattica comune, chiamata **KEYWORD**. Tuttavia, ogni parola chiave può avere la sua specifica categoria lessicale a seconda del contesto di utilizzo.

Identificatori. Nomi definiti dall'utente, come variabili, funzioni e classi. Possono essere costituiti da sequenze di lettere, numeri e simboli speciali (ad esempio, `_`), con regole che

dipendono dal linguaggio di programmazione

$$\begin{aligned}&\text{DIGIT} = [0\text{-}9]\\&\text{LETTER} = [a\text{-}zA\text{-}Z\_]\\&\{\text{LETTER}\}(\{\text{DIGIT}\} \mid \{\text{LETTER}\})\star\end{aligned}$$

Costanti. Valori specifici presenti direttamente nel codice, inclusi numeri interi (`{DIGIT}+`), numeri in virgola mobile `[+-]?[0-9]+.[0-9]*` e costanti carattere `'[^']'`

Operatori. Simboli che rappresentano operazioni matematiche, logiche o di assegnazione. Ad esempio, `+`, `-`, `*`, `/`, `==`, `&&`, ecc.

Punteggiatura. Simboli utilizzati per delimitare e strutturare il codice, come parentesi `(`, `)`, virgole `,`, punti e virgola `;`, ecc.

lessema	categoria	lessema	categoria
(	OPEN_PAREN	)	CLOSE_PAREN
[	OPEN_BRACKET	]	CLOSE_BRACKET
{	OPEN_BRACE	}	CLOSE_BRACE
+	PLUS	-	MINUS
+=	PLUS_ASSIGN	-=	MINUS_ASSIGN
:	COLON	::	SCOPE
<	LESS_THAN	<<	SHIFT_LEFT
>	GREATER_THAN	>>	SHIFT_RIGHT
.	DOT	...	ELLIPSIS
...	...		

Commenti. Parti del codice che vengono ignorate dal compilatore o interprete. Possono essere **commenti a singola linea** (`//[^\n]*\n`) o **multi-linea** (ad esempio, `/\*([^*]|\*+[^/*])*\*+/`).

Nota. La distinzione tra maiuscole e minuscole (case sensitive) o la loro equivalenza (case insensitive) è determinata dal **lexer**

Risoluzione ambiguità

Nell'analisi lessicale, si applica la regola della preferenza al lessema più lungo, che significa riconoscere la sequenza più lunga possibile di caratteri come un unico token. Ad esempio:

- $>>$ viene identificato come un unico operatore SHIFT_RIGHT, non come due operatori GREATER_THAN consecutivi.
- $+=$ viene riconosciuto come l'operatore PLUS_ASSIGN, non come una combinazione di un segno più ($+$) e un uguale ($=$).

Quando due lessemi hanno la stessa lunghezza, si stabilisce un ordine di priorità tra le espressioni regolari (RE) per decidere quale riconoscere per prima.

Attenzione. Applicare la regola della preferenza al lessema più lungo può talvolta causare errori di interpretazione. Ad esempio, nell'immagine fornita, $>>$ viene interpretato erroneamente come SHIFT_RIGHT a causa della preferenza per il lessema più lungo

Per lo standard C++03 e precedenti:

- corretto: `std::vector<std::list<int> >`
- errore (sintattico): `std::vector<std::list<int>>`

Per evitare tali errori, è spesso necessario gestire queste situazioni singolarmente, aggiungendo regole o eccezioni specifiche nell'analizzatore lessicale

Per lo standard C++11 e successivi:

- corretto: `std::vector<std::list<int> >`
- corretto: `std::vector<std::list<int>>`