4. Da espressione regolare a lexer

La costruzione di un automa a stati finiti per riconoscere un'espressione regolare è un passaggio fondamentale nella progettazione di uno lexer.

Ricordiamo che gli lexer, o analizzatori lessicali, hanno il compito di leggere il codice sorgente e suddividerlo in unità lessicali (token), come parole chiave, identificatori, operatori e simboli. Per individuare e classificare correttamente queste sequenze, gli lexer utilizzano espressioni regolari che descrivono i diversi pattern da riconoscere. Il primo passo è costruire un automa a stati finiti non deterministico (NFA) a partire dall'espressione regolare specifica per il pattern lessicale

Automi deterministici

Un automa a stati finiti deterministico (DFA) è una quintupla $M = \langle \Sigma, Q, \delta, q_0, F \rangle$ dove:

- Σ è alfabeto finito
- Q è un insieme finito degli stati dell'automa
- $\delta : Q \times \Sigma \to Q$ è una funzione di transizione
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è il sottoinsieme degli stati finali, anche detti di accettazione

Dalla funzione δ si ottiene in modo univoco la funzione $\hat{\delta} : Q \times \Sigma^* \to Q$ che ne determina la chiusura transitiva nel modo seguente

$$\begin{cases} \hat{\delta}(q, \epsilon) = q \\ \hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a) \end{cases}$$

Sia $M = \langle \Sigma, Q, \delta, q_0, F \rangle$ un DFA. Una stringa x è detta essere **accettata** da un M se $\hat{\delta}(q_0, x) \in F$. Il **linguaggio accettato** da M, denotato come $L(M)$ è quindi l'insieme di tutte le stringhe accettate da M, ovvero

$L(M) = \{x \in \Sigma^* | \hat{\delta}(q_0, x) \in F\}$

Nota. Un linguaggio è detto **regolare** se è accettato da qualche DFA

Automi non deterministici

Un automa a stati finiti non deterministico (NFA) è una quintupla $M = \langle \Sigma, Q, \delta, q_0, F \rangle$ dove Σ, Q,q_0 e F mantengono il significato visto per gli automi deterministici, mentre la funzione di transizione δ è ora definita come $\delta : Q \times \Sigma \to \mathcal{P}(Q)$, ovvero è una relazione tra stati. Anche per gli NFA dalla funzioneδsi ottiene in modo univoco la funzione $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$ nel seguente modo

$$\begin{cases} \hat{\delta}(q, \epsilon) = \{q\} \\ \hat{\delta}(q, wa) = \bigcup_{p \in \hat{\delta}(q,w)} \delta(p, a) \end{cases}$$

Sia $M = \langle \Sigma, Q, \delta, q_0, F \rangle$ un NFA. Il **linguaggio accettato** da M, denotato come $L(M)$ è quindi l'insieme di tutte le stringhe accettate da M, ovvero

$L(M) = \{x \in \Sigma^* | \hat{\delta}(q_0, x) \cap F \neq \emptyset\}$

Automi con ϵ-transazioni

In questo caso, l'idea è quella di permettere che l'automa compia transizioni di stato anche senza leggere alcun simbolo. Un NFA con ϵ-transizioni, brevemente ϵ-NFA, è una quintupla $M = \langle \Sigma, Q, \delta, q_0, F \rangle$ dove Σ, Q, q_0 e F mantengono il significato visto finora, mentre la funzione di transizioneδè ora definita come

$\delta : Q \times \{\Sigma \cup \{\epsilon\}\} \to \mathcal{P}(Q)$

La costruzione della funzione $\hat{\delta}$ richiede l'introduzione della funzione ϵ-*closure* che, applicata ad uno stato, restituisce l'insieme degli stati raggiungibili da esso (compreso sé stesso) tramite ϵ-transizioni.

La funzione $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$ può quindi essere definita nel seguente modo

$$\begin{cases} \hat{\delta}(q, \epsilon) = \epsilon\text{-closure}(q) \\ \hat{\delta}(q, wa) = \bigcup_{p \in \hat{\delta}(q,w)} \epsilon\text{-closure}(\delta(p, a)) \end{cases}$$

Il **linguaggio accettato** da M, denotato come $L(M)$ è quindi

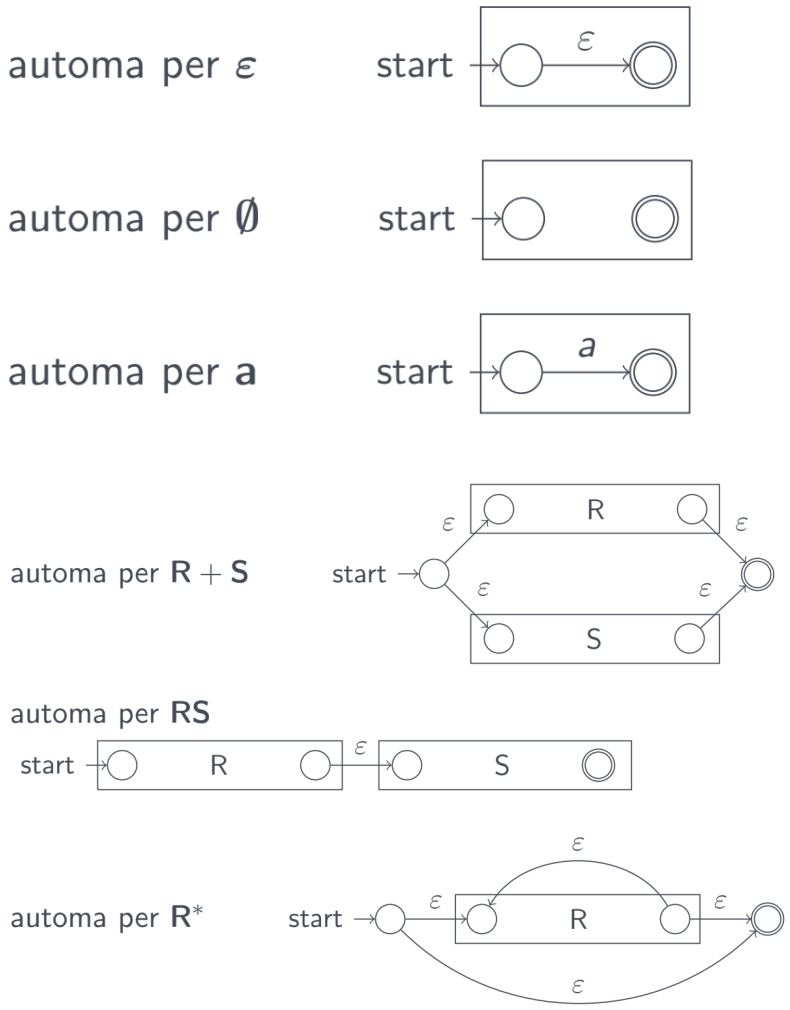
$L(M) = \{x \in \Sigma^* | \hat{\delta}(q_0, x) \cap F \neq \emptyset\}$

Nota. I linguaggi accettati dai DFA, dagli NFA e dagli ϵ-NFA coincidono.

Trasformare una RE in un ϵ-NFA

Questo è un passaggio relativamente facile da implementare in modo algoritmico, in quanto l'NFA può essere costruito combinando varie sottoespressioni (mostrate nelle immagini

sottostanti) tramite transizioni ε (algoritmo di Thompson)



La costruzione dell'NFA permette di tradurre direttamente le regole definite dalle espressioni regolari in una struttura utilizzabile per la scansione del codice.

Trasformare un ϵ-NFA in un DFA

Gli NFA non sono efficienti da utilizzare direttamente per l'analisi del codice perché richiedono una gestione complessa del non determinismo. Per questo motivo, il secondo passo è convertire l'NFA in un automa a stati finiti deterministico (DFA). Il DFA è più semplice da eseguire perché, per ogni stato, c'è una sola transizione possibile per ogni simbolo dell'alfabeto.

La costruzione del DFA avviene tramite un algoritmo basato su insiemi di stati che simula il comportamento dell'NFA e rimuove il non determinismo (costruzione a sottoinsiemi)

La **costruzione a sottoinsiemi** (o "subset construction") è un metodo utilizzato per costruire un automa a stati finiti deterministico (DFA) che simula il comportamento di un automa a stati finiti non deterministico con transizioni epsilon (ε-NFA). Di seguito è una spiegazione passo-passo di come avviene la costruzione:

Passo 1: Stato iniziale del DFA

Trova l'**epsilon-chiusura** (o epsilon-closure) dello stato iniziale dell'ε-NFA. L'epsilon-chiusura di uno stato include lo stato stesso e tutti gli stati che possono essere raggiunti da esso attraverso transizioni ε (anche multiple e in qualsiasi combinazione).

Lo stato iniziale del DFA sarà l'insieme degli stati dell'ε-NFA risultanti dall'epsilon-chiusura dello stato iniziale dell'ε-NFA.

Passo 2: Costruzione degli stati del DFA

Ogni stato del DFA corrisponde a un **insieme di stati** dell'ϵ-NFA. All'inizio, l'unico stato del DFA è l'insieme di stati ottenuto dall'epsilon-chiusura dello stato iniziale. Per ogni stato del DFA (che è un insieme di stati dell'ϵ-NFA) e per ogni simbolo dell'alfabeto:

- Considera tutti gli stati dell'ϵ-NFA nell'insieme corrente.
- Per ciascuno di questi stati, trova tutti gli stati raggiungibili leggendo il simbolo corrente.
- Trova l'epsilon-chiusura di tutti questi stati raggiungibili (unione delle epsilon-chiusure di tutti gli stati raggiunti).
- Questo insieme di stati forma un nuovo stato del DFA, a meno che non sia già stato creato in un passaggio precedente.
- Aggiungi una transizione nel DFA dallo stato corrente al nuovo stato con il simbolo corrente.

Passo 3: Stati di accettazione del DFA

Uno stato del DFA è uno stato di accettazione se **uno qualsiasi** degli stati dell'ϵ-NFA in quell'insieme è uno stato di accettazione nell'ϵ-NFA originale.

Passo 4: Ripeti la costruzione

Ripeti i passi sopra per ogni nuovo stato del DFA finché non vengono più trovati nuovi insiemi di stati (quindi nuovi stati del DFA).

Esempio.

Supponiamo di avere un ϵ-NFA con stati {0, 1, 2}, stato iniziale 0, e uno stato finale 2. Immaginiamo che:

- Lo stato 0 abbia una ϵ-transizione verso lo stato 1.
- Lo stato 1 abbia una transizione sull'alfabeto 'a' verso lo stato 2.

Costruzione del DFA.

1. **Stato iniziale del DFA:** epsilon-chiusura(0) = {0, 1}. Questo sarà lo stato iniziale del DFA, chiamiamolo A.
2. **Transizione con 'a':** Dallo stato A (0, 1), leggendo 'a', lo stato 1 può andare a 2. Quindi, lo stato risultante è 2. Troviamo la epsilon-chiusura(2) = 2. Questo sarà un nuovo stato del DFA, chiamiamolo B.
3. **Stati di accettazione:** Lo stato B contiene lo stato 2, che è uno stato finale dell'ϵ-NFA originale. Quindi, B è uno stato di accettazione del DFA.

Il DFA risultante ha due stati:

- Stato A: {0, 1}, con una transizione su 'a' verso lo stato B.
- Stato B: {2}, stato di accettazione.

Questa costruzione garantisce che il DFA risultante simulerà esattamente l'ϵ-NFA originale, ma in forma deterministica.

Minimizzazione del DFA

Una volta costruito il DFA, si passa alla minimizzazione del numero di stati per ottimizzare l’automa. Un DFA con un numero eccessivo di stati potrebbe risultare inefficiente durante la scansione del codice sorgente, aumentando i tempi di compilazione. La minimizzazione degli stati, realizzata attraverso l’algoritmo di Hopcroft, permette di ottenere un DFA il più compatto possibile, migliorando così le prestazioni dello lexer.

Ecco una spiegazione passo-passo dell'algoritmo di Hopcroft:

Concetti chiave

Prima di entrare nei dettagli dell'algoritmo, è utile comprendere un paio di concetti:

- **Partizione:** Una suddivisione degli stati del DFA in gruppi (insiemi disgiunti) chiamati blocchi.
- **Blocco:** Un insieme di stati che sono indistinguibili, ossia che non possono essere distinti in base alle loro transizioni e stati finali. Gli stati nello stesso blocco verranno eventualmente considerati equivalenti.

Passo 1: Partizione iniziale

Dividi gli stati del DFA in due blocchi:

- Uno contenente **tutti gli stati di accettazione** (stati finali).
- Uno contenente **tutti gli altri stati** (non finali).

Questa è la partizione iniziale perché gli stati di accettazione e quelli non di accettazione sono sicuramente distinguibili tra loro.

Passo 2: Gestione delle partizioni

Crea un insieme di blocchi in attesa di essere suddivisi, chiamato "lavoro in sospeso". Inizialmente, questo insieme contiene i due blocchi della partizione iniziale (stati finali e non finali).

Continua il processo finché "lavoro in sospeso" non è vuoto:

- Prendi un blocco P dall'insieme "lavoro in sospeso".
- Per ogni simbolo dell'alfabeto a:
  - Trova gli stati che transitano verso gli stati in P con il simbolo a. Questo insieme è chiamato il **pre-immagine** di P tramite a.
  - Per ogni blocco B della partizione corrente, verifica come il pre-immagine di P tramite a si interseca con B. Se il pre-immagine di P divide B in due insiemi non vuoti B_1 e B_2, allora:
    1. Sostituisci B con i due sottoinsiemi B_1 e B_2 nella partizione.
    2. Aggiungi uno dei due sottoinsiemi (B_1 o B_2) all'insieme "lavoro in sospeso" (se uno dei due sottoinsiemi era già presente in "lavoro in sospeso", aggiorna con il nuovo insieme più piccolo).

Questo passaggio assicura che gli stati nel DFA siano continuamente suddivisi in blocchi più piccoli finché non possono più essere distinti l'uno dall'altro.

Passo 3: Costruzione del DFA minimizzato

Una volta che l'insieme "lavoro in sospeso" è vuoto, la partizione risultante contiene blocchi di stati che sono **indistinguibili** tra loro.

Ogni blocco della partizione rappresenta un singolo stato nel DFA minimizzato.

Definisci le transizioni tra i nuovi stati del DFA minimizzato in base alle transizioni tra gli stati originali:

- Se c'è una transizione da uno stato del blocco B_1 a uno stato del blocco B_2 nel DFA originale con un simbolo a, allora nel DFA minimizzato ci sarà una transizione dallo stato corrispondente a B_1 a quello corrispondente a B_2 con il simbolo a.

L'**insieme degli stati finali** nel DFA minimizzato è dato dai blocchi della partizione che contengono almeno uno stato di accettazione del DFA originale.

Lo **stato iniziale** del DFA minimizzato è il blocco della partizione contenente lo stato iniziale del DFA originale.

Esempio.

Supponiamo di avere un DFA con gli stati $\{q_0, q_1, q_2, q_3\}$
Stato iniziale: q_0
Stati finali: q_2 e q_3
Transizioni:

- q_0 con 'a' va a q_1, con 'b' va a q_2.
- q_1 con 'a' va a q_0, con 'b' va a q_3.
- q_2 con 'a' va a q_3, con 'b' va a q_0.
- q_3 con 'a' va a q_2, con 'b' va a q_1.

Passo 1: Partizione iniziale

- Stati finali: $\{q_2, q_3\}$.
- Stati non finali: $\{q_0, q_1\}$.

Passo 2: Partizione successiva

- Esamina le transizioni degli stati e dividi ulteriormente i blocchi in base alle loro transizioni con gli stessi simboli.
- Supponendo che dopo alcune iterazioni non siano più possibili suddivisioni, i blocchi risultanti saranno, ad esempio: $\{q_0, q_1\}$ e $\{q_2, q_3\}$.

Passo 3: Costruzione del DFA minimizzato

- Il DFA minimizzato avrà 2 stati corrispondenti ai blocchi trovati.
- Definiamo le transizioni tra questi nuovi stati in base alle transizioni degli stati originali.

L'algoritmo di Hopcroft ha una complessità di $O(n \log n)$, dove n è il numero di stati del DFA, rendendolo uno degli algoritmi più efficienti per la minimizzazione dei DFA. In sintesi, l'algoritmo divide gli stati del DFA in blocchi e li suddivide ulteriormente finché non può più farlo, garantendo che ogni stato nel DFA minimizzato rappresenti un insieme di stati equivalenti nel DFA originale.

Trasformare un DFA in uno lexer

Infine, dopo aver costruito e ottimizzato il DFA, si genera il codice dello lexer. In questa fase, vengono specificate le azioni da intraprendere quando vengono riconosciute determinate sequenze di input, associando così i token alle varie parti del codice sorgente.