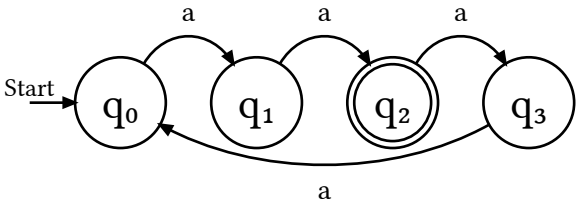
Soluzione Linguaggi Totale

2024-06-04

NOTA: Questa è una soluzione proposta da me, non è detto che sia giusta. Se trovate errori segnalateli o meglio correggeteli direttamente :)

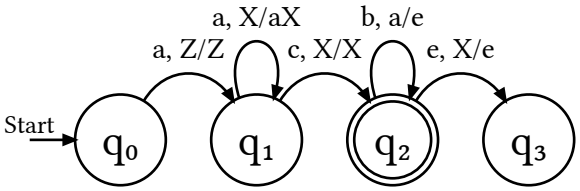
1. Per dimostrare che le definizioni di grammatiche libere **sono equivalenti**, osserviamo che:
   - Ogni grammatica di tipo (b) è pure una grammatica di tipo (a).
   - Per ogni grammatica di tipo (a) posso trovare una equivalente grammatica di tipo (b). Infatti utilizzando l'algoritmo per la rimozione delle produzioni ε, data G di tipo (a), genero una G' tale che $L(G') = L(G) \setminus \{\varepsilon\}$ e le cui produzioni non sono epsilon. Per ottenere una del tutto equivalente, nel caso in cui $\varepsilon \in L(G)$, basta modificare $G' = (NT, T, S, R)$ con l'aggiunta di un nuovo simbolo $S' \to \varepsilon \mid S$ e $G'' = (NT \cup \{S'\}, T, S', R \cup \{S' \to \varepsilon, \mid S\})$. Tale che G'' è di tipo (b).

2. Costruisco l'NFA più semplice che riconosce il linguaggio:



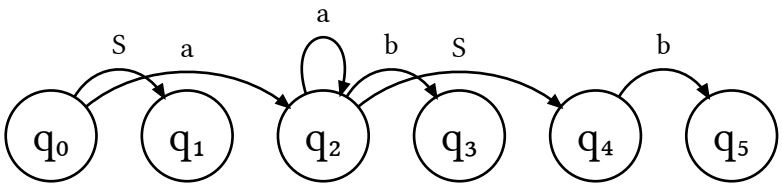
   Si può direi che è un DFA minimo e genera $L = \{a^2, a^6, a^{10}, ...\}$ con regex $\text{aa}(\text{aaaa})^*$ ma non è richiesto.

3. Non riesco a fare le ε in questo pacchetto, assumete $e = \varepsilon$:



4. Per il linguaggio $L = \{a^n b^n \mid n \geq 1\}$ una grammatica possibile è:
   1. $S \to \text{ab}$
   2. $S \to \text{aSb}$

   Costruiamo l'automa $LR(0)$. Non so come fare stati grandi con tutti gli items in typst quindi scrivo l'automa e poi definisco gli stati a parte:



   Items:
   - q0:
     $S' \to .S$
     $S \to .\text{ab}$
     $S \to .\text{aSb}$
   - q1: $S' \to S.$

- q2:
  $S \to$ a.b
  $S \to$ a.Sb
  $S \to$ .ab
  $S \to$ .aSb
- q3: $S \to$ ab.
- q4: $S \to$ aS.b
- q5: $S \to$ aSb.

Costruiamo la tabella $LR(0)$:

	a	**b**	**$**	**S**
0	S2			G1
1			Acc	
2	S2	S3		G4
3	R1	R1	R1	
4		S5		
5	R2	R2	R2	

Non ci sono conflitti, quindi la grammatica è $LR(0)$. Discutiamo il comportamento del parser sugli input:
$(0, \varepsilon, \text{aabb\$})$
$(02, a, \text{abb\$})$
$(022, \text{aa}, \text{bb\$})$
$(0223, \text{aab}, \text{b\$})$ Facciamo la reduce: rimuoviamo "23" e [2, S] -> G4
$(024, \text{aS}, \text{b\$})$
$(0245, \text{aSb}, \$)$
$(01, S, \$)$ Accept!

$(0, \varepsilon, \$)$ Errore, cella bianca

5. Direi che il testo è sbagliato e il passaggio dei paramentri è per nome e per riferimento. Per nome basta ricopiare il codice al posto della chiamata (con le opportune accortezze):

```
{
  int x = 1;
  int y = 5;
  int z = 10;
  void pippo ( name int y , reference int z ){
    z = x + y + z;
  }

  {
    int x = 20;
    int y = 30;
    int z = 50;
    //pippo (x++ , x);
    // x_ref sarebbe riferimento a x che vale 20. x_top è la x sopra la
    // funzione pippo per scope statico.
    // In C la segunte istruzione è undefined behavior. Assumiamo che si
    // valuti strettamente da sinistra a destra.
    x_ref = x_top + (x_ref++) + x_ref; // x_ref = 1 + 20 + 21 = 42
```

```
    //pippo (x++ , x);
    x_ref = x_top + (x_ref++) + x_ref; // x_ref = 1 + 42 + 43 = 86
    write (x); // 86
  }

  write (x); // 1
}
```

In conclusione il programma stampa **86** e **1**

6. Assumiamo che la variabile letta all'esterno sia `n`:

```
int f(int x, function g()) {
  if (x == 0) {
    g();
    return 0;
  } else {
    return f(x - 1, g()) + 1; // Evito che f possa essere tail recursive
  }
}
{
  void h() {
    // Codice generico
  }

  g(n, h)
}
```

L'idea è di usare `f()` per creare `n` record di attivazione e poi far chiamare `h()` a `f()`.

7. È da pensare come javascript. Quindi `h({})` ritorna `{c: {e: 4}}`. Con questa logica `x` risulta essere: `x = {a: 1, b: {c: {e: 4}, d: {a: 5, b: {c: {a: 6 , e: 7}, d: {e: 8}}}} }`. Quindi:
   - **I1**: Errata in quanto `x` ha il campo `x.b.c` ma non contiene `b`.
   - **I2**: Errata, `x` non ha il capo `c`.
   - **I3**: Giusta
   - **I4**: Errata
   - **I5**: Giusta

8. Dal seguente programma in C++ possiamo vedere che stampa **2 3 5 7 11**:

```
#include <exception>
#include <iostream>
#include <list>

class Z : std::exception {};

void d(int i, int n) { if (i % n == 0) throw Z(); }

void b(std::list<int> &s, int n);

class X : std::exception {};
void c(std::list<int> &s, int i, int n) {
  try { b(s, n); } catch (X) { }
  s.push_front(i); // Come add() del testo
}

void b(std::list<int> &s, int n) {
  if (s.empty()) { throw X(); }
```

```
  int i = s.front(); s.pop_front(); // Come remove() del testo
  try {
    d(i, n);
    c(s, i, n);
  } catch (Z) {
    b(s, n);
  }
}

class Y : std::exception {};
void a(std::list<int> &s) {
  if (s.empty()) {
    throw Y();
  }
  int n = s.front(); s.pop_front(); // Come remove() del testo

  try { b(s, n); } catch (X) { }
  try { a(s); } catch (Y) { }
  s.push_front(n); // Come add() del testo
}

int main() {
  std::list<int> l = {2, 8, 6, 3, 5, 12, 7, 11, 14, 10};
  a(l);
  for (auto &e : l) {
    std::cout << e << " ";
  }
  std::cout << std::endl;
}
```

L'idea è la seguente: l'insieme delle classi partendo dal primo elemento della lista rimuovono tutti i multipli di quell'elemento. Quando non ci sono più multipli si passa all'elemento sucessivo. Durante il processo tutti gli elementi vengono tolti dalla lista. Quando finalmente la lista si svuota, vengono reinseriti tutti gli elementi in testa.

La funziona `b()` è quella responsabile di rimuovere gli elementi. La funziona `d()` controlla se `i` è multiplo di `d` e nel caso lancia `Z` che evita che venga chiamata `c()` che è la funzione responsabile a "salvare" i numeri che poi verranno reinseriti nella lista.