Domande Informatica Teorica

1. Cos'è un problema di decisione?

   Solution: Un problema di decisione è una domanda a cui rispondere *Sì* o *No*.
   Formalmente, è costituito da 3 elementi:

   - Nome del problema
   - Istanza degli oggetti considerati
   - Domanda, ovvero proprietà che gli oggetti possono o meno soddisfare

2. Come dimostrare l'esistenza di problemi non decidibili, senza mostrarne un esempio?

   Solution: Per studiare la decidibilità di un problema Π ci sono due possibili approcci:

   - Trovare un programma P_Π che calcola la funzione soluzione

     $$\Phi_\Pi : D \to \{0,1\} \quad \text{t.c.} \quad \Phi_\Pi(x) = \begin{cases} 1 & \text{se } p(x) \\ 0 & \text{se } \neg p(x) \end{cases}$$

     di conseguenza $\Phi_\Pi \in \mathcal{T}$
   - Se $\Phi_\Pi \in \mathcal{T}$, allora esiste un programma che la calcola

   Di conseguenza, i problemi risolvibili PROG sono isomorfi alle funzioni calcolabili, quindi numerabili, mentre tutti i possibili problemi sono rappresentati dalle funzioni da $\mathbb{N}$ a $\mathbb{N}$ (dato quanto già visto), isomorfe a $\mathbb{N}_\perp^\mathbb{N}$, quindi:

   $$\text{DATI} \sim \text{PROG} \sim \mathbb{N} \nsim \mathbb{N}_\perp^\mathbb{N}$$

   Per dimostrare che DATI $\sim \mathbb{N}$: serve una funzione tale che permetta una biezione tra dati e $\mathbb{N}$, come la funzione coppia di Cantor

   $$\langle \_, \_ \rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}^+$$

   (banalmente estendibile a tutto $\mathbb{N}$).

   Per dimostrare che PROG $\sim \mathbb{N}$: una volta definito un sistema di calcolo e i relativi comandi, si può mostrare una codifica per questi che porta a una biezione con $\mathbb{N}$.

   Alternativamente, un problema di decisione può essere comparato al riconoscimento di un linguaggio $L \subseteq \Sigma^*$, per un alfabeto Σ di conseguenza:

- Il numero possibile di linguaggi è $P(\Sigma^*) \sim \mathbb{R}$
- I sistemi di calcolo sono in quantità PROG $\sim \mathbb{N}$

Per forza devono esistere linguaggi non decidibili.

3. Il problema dell'arresto: definizione e dimostrazione.

Solution: Definizione del problema dell'arresto:

- Nome: AR
- Istanza: $x, y \in \mathbb{N}$
- Domanda: $\varphi_y(x) \downarrow$?

Teorema 0.1. *AR è indecidibile.*

Dimostrazione. Assumiamo per assurdo che AR sia decidibile, allora esiste una funzione soluzione

$$\Phi_{\text{AR}}(x, y) = \begin{cases} 0 & \text{se } \varphi_y(x) \uparrow \\ 1 & \text{se } \varphi_y(x) \downarrow \end{cases}$$

Valutando il caso in cui $x = y$

$$\Phi_{\text{AR}}(x, x) = \begin{cases} 0 & \text{se } \varphi_x(x) \uparrow \\ 1 & \text{se } \varphi_x(x) \downarrow \end{cases}$$

Visto che $\Phi_{\text{AR}} \in \mathcal{T}$, anche la funzione

$$f(x) = \begin{cases} 0 & \text{se } \Phi_{\text{AR}}(x) = 0 \equiv \varphi_x(x) \uparrow \\ \varphi_x(x) + 1 & \text{se } \Phi_{\text{AR}}(x) = 1 \equiv \varphi_x(x) \downarrow \end{cases} \in \mathcal{T}$$

Sia $\alpha \in \mathbb{N}$ la codifica di A tale che $\varphi_\alpha = f$. Valutiamo φ_α in α:

$$\varphi_\alpha(\alpha) = \begin{cases} 0 & \text{se } \varphi_\alpha(\alpha) \uparrow \\ \varphi_\alpha(\alpha) + 1 & \text{se } \varphi_\alpha(\alpha) \downarrow \end{cases}$$

Ma tale funzione non può esistere:

- Nel primo caso $\varphi_\alpha(\alpha) = 0$ se $\varphi_\alpha(\alpha) \uparrow$, ma è una contraddizione
- Nel secondo caso $\varphi_\alpha(\alpha) = \varphi_\alpha(\alpha) + 1$, ma tale relazione non vale per nessun naturale

Siamo a un assurdo, AR è indecidibile.

□

4. Sistemi di calcolo visti in Teoria della Calcolabilità.

Solution: I sistemi di calcolo visti sono:

- Sistema RAM: infiniti registri, R_0 contiene l'output, R_1 l'input, si ha un program counter L, le istruzioni sono
  - Incremento: $R_k \leftarrow R_k + 1$
  - Decremento: $R_k \leftarrow R_k \dot{-} 1$
  - Salto condizionato: `if` $R_k = 0$ `goto` m, con $m \in \{1, \ldots, |P|\}$
- Sistema WHILE: 21 registri, x_0 output, x_1 input, sono presenti dei comandi base:
  - $x_k := x_j + 1$
  - $x_k := x_j \dot{-} 1$
  - $x_k := 0$

  e dei comandi definiti induttivamente:
  - Comando composto

    `begin` $C_1, \ldots, C_n$ `end`

    dove ogni C_i è un qualsiasi comando
  - Comando while

    `while` $x_k \neq 0$ `do` C

    dove C è un qualsiasi comando

  Di conseguenza, un programma WHILE è un comando composto

5. Approfondimento su RAM (struttura, istruzioni, stato prossimo, computazione)

Solution: Il sistema RAM permette infiniti registri, tra i quali R_0 contiene l'output e R_1 l'input, si ha inoltre un program counter L per tenere traccia dell'istruzione da eseguire. Un programma P è un insieme ordinato di istruzioni.

Le istruzioni sono:

- Incremento: $R_k \leftarrow R_k + 1$
- Decremento: $R_k \leftarrow R_k \dot{-} 1$

- Salto condizionato: `if` $R_k = 0$ `goto` m, con $m \in \{1, \ldots, |P|\}$

Ogni istruzione fa passare la macchina da uno stato a un altro; la semantica operazione di un'istruzione è formata dalla coppia degli stati prima e dopo l'istruzione.

La computazione del programma P è una sequenza di stati $\mathcal{S}_i$, infinita se non termina, altrimenti si ha uno stato finale $\mathcal{S}_{fin}$ in cui viene posto in R_0 il risultato della computazione.

Lo stato è una funzione

$$\mathcal{S} : \{L, R_i\} \to \mathbb{N}$$

ovvero, che dato un registro e un valore del program counter L, restituisce il contenuto del registro.

Uno stato finale $\mathcal{S}_{fin}$ è un qualsiasi stato tale che $\mathcal{S}(L) = 0$.

Lo stato iniziale è tale che

$$\mathcal{S}_{init}(R_i) = \begin{cases} 1 & \text{se } R_i = L \\ n & \text{se } R_i = R_1 \\ 0 & \text{altrimenti} \end{cases}$$

Per definire l'esecuzione del programma si usa la funzione stato prossimo

$$\delta : \text{STATI} \times \text{PROG} \to \text{STATI}_\perp$$

tale che

$$\delta(\mathcal{S}, P) = \mathcal{S}'$$

Dove $\mathcal{S}$ rappresenta lo stato in seguito all'esecuzione del comando P.

La funzione è tale che:

- Se $\mathcal{S}(L) = 0$, $\mathcal{S}' = \perp$ in quanto l'esecuzione è terminata
- Se $\mathcal{S}(L) > |P|$ non si ha una terminazione esplicita, quindi

$$\mathcal{S}'(R) = \begin{cases} 0 & \text{se } R = L \\ \mathcal{S}(R_i) & \text{altrimenti} \end{cases}$$

- Se $1 \leq \mathcal{S}(L) \leq |P|$, si considera l'istruzione $\mathcal{S}(L)$-esima:
  - incremento/decremento su R_k:

$$\mathcal{S}'(R) = \begin{cases} \mathcal{S}(R) + 1 & \text{se } R = L \\ \mathcal{S}(R) \pm 1 & \text{se } R = R_k \\ \mathcal{S}(R) & \text{altrimenti} \end{cases}$$

– salto condizionato su R_k

$$\mathcal{S}'(R) = \begin{cases} m & \text{se } R = L \wedge R_k = 0 \\ \mathcal{S}(L) + 1 & \text{se } R = L \wedge R_k \neq 0 \\ \mathcal{S}(R) & \text{altrimenti} \end{cases}$$

L'esecuzione di un programma genera una sequenza di stati, definita secondo la funzione δ.

6. Struttura e istruzioni WHILE.

Solution: Il sistema di calcolo WHILE usa esattamente 21 registri ed è strutturato, quindi non necessita di program counter. Sono presenti istruzioni base e istruzioni definite induttivamente. Comandi base:

- $x_k := x_j + 1$
- $x_k := x_j \dot{-} 1$
- $x_k := 0$

Comandi induttivi:

- Comando composto

  $$\texttt{begin } C_1; \ldots; C_n \texttt{ end}$$

  dove ogni C_i è un qualsiasi tipo di comando
- Comando while

  $$\texttt{while } x_k \neq 0 \texttt{ do } C$$

  dove C è un qualsiasi comando

Per dimostrare proprietà di un programma $P \in W$-PROG la si verifica prima sui comandi base, poi induttivamente su comando composto e while.

Dati comando da eseguire e stato corrente, la funzione stato prossimo restituisce lo stato successivo:

$$[\![\_]\!](\_) : W\text{-COM} \times W\text{-STATI} \to W\text{-STATI}_\perp$$

Si può definire la semantica induttivamente, a partire dagli assegnamenti:

$$[\![x_k := x_j \pm 1]\!](\underline{x}) = \begin{cases} x_i & \text{se } i \neq k \\ x_j \pm 1 & \text{se } i = k \end{cases}$$

$$[\![x_k := 0]\!](\underline{x}) = \begin{cases} x_i & \text{se } i \neq k \\ 0 & \text{se } i = k \end{cases}$$

Mentre per i comandi induttivi, partendo dal comando composto:

$$[\![\texttt{begin } C_1; \ldots; C_n \texttt{ end}]\!](\underline{x}) = ([\![C_n]\!] \circ \cdots \circ [\![C_1]\!])(\underline{x})$$

dove i comandi C_i sono noti per I.H.

Per il comando while:

$$[\![\texttt{while } x_k \neq 0 \texttt{ do } C]\!](\underline{x}) = \begin{cases} [\![C]\!]^e(\underline{x}) & \text{se } \exists e \text{ t.c. } [\![C]\!]^e(\underline{x})[k] = 0 \\ \bot & \text{altrimenti} \end{cases}$$

Di conseguenza, dato che un programma while $W \in W$-PROG è un comando composto, si può rappresentare la semantica del programma come

$$\Psi_W(n) = \text{Pro}_0^{21}([\![W]\!](w\text{-}in(n)))$$

dove $w\text{-}in(n)$ restituisce lo stato iniziale della macchina con input n.

7. Relazione tra RAM e WHILE, con dimostrazione che il secondo è contenuto nel primo e viceversa.

Solution: Si può dimostrare che le funzioni calcolabili dai due sistemi sono le stesse, ovvero che $F(\text{RAM}) = F(\text{WHILE})$. Per dimostrarlo si deve mostrare che $F(\text{WHILE}) \subseteq F(\text{RAM})$ e $F(\text{RAM}) \subseteq F(\text{WHILE})$.

Dati due sistemi di calcolo $\mathcal{C}_1$ e $\mathcal{C}_2$, si definisce traduzione una funzione che associa i programmi di uno ai programmi dell'altro, con le proprietà di programmabilità, correttezza e completezza, rispetto ai programmi presenti nei due insiemi. Se esiste un traduttore da $\mathcal{C}_1$ a $\mathcal{C}_2$ allora $F(\mathcal{C}_1) \subseteq F(\mathcal{C}_2)$, dato che per ogni programma P_1 del primo sistema di calcolo esiste (per completezza) $P_2 = t(P_1)$ nel secondo con la stessa semantica (per correttezza).

Per dimostrare che $F(\text{WHILE}) \subseteq F(\text{RAM})$, costruiamo un compilatore $C : W\text{-PROG} \to$ PROG. Si può definire induttivamente il compilatore, in quanto WHILE è definito induttivamente. I 21 registri vengono mappati nei primi 21 RAM, dato che sono infiniti non ci sono problemi.

Passo base: gli assegnamenti

- $x_k := 0$, si itera sul registro da azzerare finché è diverso da 0 (usando $R_{21} = 0$ come condizione di coda del ciclo)
- $x_k = x_j \pm 1$, se $k = j$ è banale, altrimenti bisogna
  - Spostare il valore di R_j in R_{22}, azzerandolo
  - Azzerare R_k
  - Rigenerare R_j e R_k a partire da R_{22}
  - Effettuare l'incremento/decremento effettivo

Passo induttivo:

- Il comando composto è una composizione di comandi noti per I.H.
- Il comando while si può compilare come un ciclo che esegue il comando, noto per I.H., finché $R_k \neq 0$

Il compilatore così definito è programmabile, compila ogni $W \in W$-PROG e mantiene la semantica, di conseguenza

$$F(\text{WHILE}) \subseteq F(\text{RAM})$$

Per mostrare che $F(\text{RAM}) \subseteq F(\text{WHILE})$ vogliamo scrivere un interprete I_W che prende in input un programma $P \in \text{PROG}$ e $x \in \mathbb{N}$ e restituisce l'esecuzione di P su x, ovvero $\varphi_P(x)$. Non crea nulla di intermedio, si limita a eseguire il programma.

Dovendo prendere due input (P e x), bisogna condensarli tramite Cantor, quindi, con $n = cod(P)$, l'input diventa $\langle x, n \rangle$. Di conseguenza:

$$\forall x, n \in \mathbb{N},\ \Psi_{I_W}(\langle x, n \rangle) = \varphi_n(x) = \varphi_P(x)$$

Come salvare lo stato della macchina nell'interprete? Un problema sono i registri, in RAM possono essere infiniti, ma non ne verranno mai usati veramente infiniti, quindi il contenuto di questi può essere rappresentato tramite una lista $\langle R_0, \ldots, R_n \rangle$.

L'interprete I_W salva lo stato della macchina nel modo:

- $x_0 \leftarrow \langle R_0, \ldots, R_n \rangle$
- $x_1 \leftarrow L$
- $x_2 \leftarrow y$, dato su cui lavora P

- $x_3 \leftarrow n$, "listato" del codice
- $x_4 \leftarrow I$, istruzione attuale, prelevata dal listato grazie al PC

Inizialmente l'input $\langle x, n \rangle$ si trova in x_1. L'interprete deve quindi solamente estrarre, nell'ordine stabilito dal PC, le istruzioni dal listato, decodificare ed eseguirle, finché il PC è diverso da zero, ovvero $x_1 \neq 0$.

Avendo l'interprete, si può costruire un compilatore $C : \text{PROG} \to W\text{-PROG}$ semplicemente chiamando l'interprete dopo aver posto in x_1 l'input $\langle x_1, x_2 \rangle$, dove x_1 contiene l'input di P e x_2 contiene $cod(P)$.

Abbiamo quindi trovato un compilatore programmabile, completo e corretto da RAM a WHILE, mostrando che

$$F(\text{RAM}) \subseteq F(\text{WHILE})$$

Di conseguenza

$$F(\text{RAM}) = F(\text{WHILE})$$

8. Differenza tra compilatore e interprete.

Solution: Un compilatore prende in input un programma e lo trasforma in un altro in un sistema di calcolo destinazione, mantenendone la semantica.

Un interprete è una funzione che prende come argomenti un programma e un input e simula l'esecuzione del programma passo passo, restituendo l'esecuzione del programma originale sull'input fornito.

9. Come passare un programma RAM all'interprete.

Solution: L'interprete accetta in input la codifica del programma (quindi un numero, dato che PROG $\sim \mathbb{N}$) e ricava l'istruzione da prelevare grazie al valore del PC, per poi decodificarla.

10. Perché è impossibile che il destro di una coppia di Cantor sia 0, a meno che non sia l'ultima istruzione codificata.

Solution: Dopo aver applicato iterativamente Cantor su una lista $\langle x_1, \langle \dots \langle x_n, 0 \rangle \dots \rangle \rangle$, quando vengono estratti i valori, come sin verrà estratto l'ultimo valore inserito, come des rimarrà la codifica di tutti gli altri valori della lista, che quindi potrà essere 0 solo nel caso in cui questa è l'ultima istruzione codificata.

11. Gerarchia classi di complessità e descrizione.

Solution: Le classi di complessità sono

$$L \subseteq P \subseteq NP \subseteq \text{EXPTIME} \subseteq \mathscr{P}$$

E sono definite come:

- $L = \text{DSPACE}(\log n)$
- $P = \text{DTIME}\left(n^k\right)$
- $NP = \text{NTIME}\left(n^k\right)$
- $\text{EXPTIME} = \text{DTIME}\left(2^{n^k}\right)$
- $\mathscr{P}$ tutte le funzioni calcolabili

12. Descrizione DTM per calcolare lo spazio.

Solution: Data la DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ si definisce $S(x)$ il numero di celle occupate sul nastro di memoria durante la computazione di M su x.

La complessità in spazio viene conseguentemente definita come

$$s(x) = \max\{S(x) \mid x \in \Sigma^* \wedge |x| = n\}$$

ovvero come il massimo numero di celle usate per il calcolo di un input, per ogni taglia di input.

Ma in questo modo viene considerato anche lo spazio dedicato all'input, non permettendo complessità sublineari. Per risolvere, si aggiunge un altro nastro dedicato all'input, con dei terminatori $\notin \Sigma$. Si deve modificare la funzione di transizione per gestire il moto delle due testine e permettere di leggere sia da nastro di input che di lavoro.

Per ogni $x \in \Sigma^*$, $S(x)$ è ora dato solo dal numero di celle usate solamente sul nastro di lavoro, non considerando così l'interferenza dovuta all'input.

Il linguaggio $L \in \Sigma^*$ è riconosciuto in spazio deterministico $f(n)$ se e solo se esiste una DTM M tale che $L = L_M$ e $s(n) \leq f(n)$.

13. Com'è definita la computazione in una DTM.

Solution: La computazione è una sequenza di mosse (passo che, dato lo stato corrente e simbolo letto, porta a un nuovo stato, con eventuale scrittura e spostamento) dettate dalla funzione di transizione.

La funzione di transizione è definita come

$$\delta : Q \times \Gamma \to Q \times (\Gamma \setminus \{blank\}) \times \{-1, 0, 1\}$$

14. Qual è la condizione di terminazione in una DTM.

Solution: Una DTM termina quando entra in uno stato finale, ovvero con $\delta(q, \gamma) = \perp$, oppure quando la funzione di transizione non è definita (si tratta di una funzione parziale).

15. Definizione ricorsive primitive partendo da ELEM.

Solution: L'insieme ELEM contiene solo le operazioni di successore, proiettore e azzeramento. Aggiungendo l'operatore di composizione

$$\text{Comp}(h, g_1, \ldots, g_k) : \mathbb{N}^n \to \mathbb{N} \mid \text{Comp}(h, g_1, \ldots, g_k)(\underline{x}) = h(g_1(\underline{x}) \circ \cdots \circ g_n(\underline{x}))$$

e l'operatore di ricorsione primitiva

$$\text{RP}(h, g) = f(\underline{x}, y) = \begin{cases} g(\underline{x}) & \text{se } y = 0 \\ h(f(\underline{x}, y), y - 1, \underline{x}) & \text{se } y > 0 \end{cases}$$

per poi chiudere l'insieme rispetto alle due operazioni, si ottiene l'insieme RICPRIM

$$\text{ELEM}^{\{\text{Comp,RP}\}} = \text{RICPRIM}$$

16. Definizione ricorsive parziali.

Solution: A partire da RICPRIM (definito sopra), per aggiungere all'insieme delle funzioni calcolabili le ricorsive parziali si usa l'operatore di minimalizzazione:

$$\mathrm{MIN}(f)(\underline{x}) = \mu_y(f(\underline{x}, y) = 0) = \begin{cases} y & \text{se } f(\underline{x}, y) = 0 \wedge (\forall y' \leq y, f(\underline{x}, y) \downarrow \wedge f(\underline{x}, y') \neq 0) \\ \perp & \text{altrimenti} \end{cases}$$

Informalmente, restituisce il più piccolo valore di y che azzera $f(\underline{x}, y)$, ovunque precedentemente definita su y'. Questo operatore permette le presenza di funzioni parziali e chiudere RICPRIM rispetto alla minimalizzazione permette di ottenere

$$\mathrm{ELEM}^{\{\mathrm{Comp,RP,MIN}\}} = \mathscr{P}$$

ovvero l'insieme delle funzioni ricorsive parziali.

17. RICPRIM = F(FOR)

Solution: Tutte le funzioni in RICPRIM hanno inizio e fine ben definiti, sono tutte funzioni totali (ma non rappresenta tutte le funzioni totali), quindi sulla base di questo si può definire il costrutto FOR che si serve di una variabile di controllo che parte da un valore ed arriva a un valore limite.

Le funzioni calcolabili tramite FOR sono funzioni totali con numero di iterazioni noto, esattamente come per RICPRIM.

18. Relazione tra $\mathscr{P}$ e F(WHILE).

Solution: Per confrontare $\mathscr{P}$ e F(WHILE) si può cominciare dal domandarsi se $\mathscr{P} \subseteq F$(WHILE). L'insieme $\mathscr{P}$ può essere visto come definito induttivamente, ovvero

- Se $f \in \mathrm{ELEM}$ allora $f \in \mathscr{P}$
- Se $h, g_1, \ldots, g_k \in \mathscr{P}$ allora $\mathrm{Comp}(h, g_1, \ldots, g_k) \in \mathscr{P}$
- Se $h, g \in \mathscr{P}$ allora $\mathrm{RP}(h, g) \in \mathscr{P}$
- Se $f \in \mathscr{P}$, allora $\mathrm{MIN}(f) \in \mathscr{P}$

Di conseguenza, proviamo per induzione strutturale su $\mathscr{P}$ che le funzioni sono while-programmabili:

- Passo base: le funzioni elementari sono semplicemente programmabili tramite while
- Passi induttivi:
  - Per $\text{Comp}(h, g_1, \ldots, g_k) \in \mathscr{P}$: si tratta di una composizione di istruzioni base, programmabili per I.H.
  - Per $\text{RP}(h, g) = f(\underline{x}, y)$: si usa un comando while e una variabile di controllo per iterare un singolo comando, programmabile per I.H., finché la variabile di controllo non ha raggiunto il valore di y
  - Per $\text{MIN}(f) \in \mathscr{P}$: un ciclo while può incrementare il valore di 1 finché non azzera la funzione, ottenendo la stessa semantica della minimalizzazione

Quindi $\mathscr{P} \subseteq F(\text{WHILE})$.

Per $F(\text{WHILE}) \subseteq \mathscr{P}$: si può rappresentare la semantica di un programma $W \in W\text{-PROG}$ con

$$\Psi_W = \text{Pro}_0^{21}([\![W]\!](w\text{-}in(x)))$$

Quindi, sapendo che l'operatore di proiezione è $\in \mathscr{P}$, se la funzione di stato prossimo è ricorsiva parziale abbiamo verificato l'inclusione.

Definiamo la funzione numero prossimo f_C che applica Cantor all'array degli stati; si può passare da $[\![C]\!](x)$ a $f_C(x)$ usando solo funzioni ricorsive parziali. Verifichiamo che f_C sia ricorsiva parziale per induzione strutturale:

- Passo base:
  - $C \equiv x_k := 0$

    $$f_C = \left[\text{Pro}_0^{21}(x), \ldots, 0, \ldots, \text{Pro}_2^{21}0(x)\right]$$

    con lo 0 in posizione k
  - $C \equiv x_k := x_j \pm 1$

    $$f_C = \left[\text{Pro}_0^{21}(x), \ldots, \text{Pro}_j^{21}(x) + 1, \ldots, \text{Pro}_2^{21}0(x)\right]$$

    Avendo usato solo funzioni $\in \mathscr{P}$, i due comandi sono $\in \mathscr{P}$
- Passo induttivo:
  - $C \equiv \texttt{begin } C_1; \ldots; C_n; \texttt{ end}$

    $$f_C = f_{C_n}(\ldots f_{C_1}(x) \ldots)$$

- $C \equiv$ `while` $x_k \neq 0$ `do` C_b

  $$f_C(x) = f_{C_b}^{e(x)}(x) \text{ con } e(x) = \mu_y(\text{Pro}(k, f_{C_b}^y(x)))$$

  $e(x)$ non è costante, ma si può definire $T(x, y) = f_{C_b}^y(x)$, facilmente implementabile tramite ricorsione primitiva, quindi $\in \mathscr{P}$, facendo diventare $e(x)$ una minimalizzazione di funzioni $\in \mathscr{P}$, quindi $e(x) \in \mathscr{P}$.

Quindi anche questi sono $\in \mathscr{P}$

Abbiamo provato anche che $F(\text{WHILE}) \in \mathscr{P}$.

Di conseguenza, in totale

$$\mathscr{P} = F(\text{WHILE})$$