الگوریتم یک جواب متنهای به تعداد نامتناهی سوال است.

مدارهای بولینی (Boolean circuit) می‌توانند فقط توابع متناهی (محدود) را محاسبه کنند. بنابراین آن‌ها برای تعریف الگوریتم به عنوان یک تعریف واحد (single recipe) که توابع نامتناهی (نامحدود) را محاسبه کنند، مناسب نیستند.

ما در اینجا به سراغ اینکه چه چیزی را (What) باید محاسبه کنیم می‌رویم و اینکه چجوری (How) آن را محاسبه کنیم را در فصل‌های بعد بررسی می‌کنیم.

6.1 – FUNCTIONS WITH INPUT OF UNBOUNDED LENGTH :

تعریف تابع XOR با ورودی نامحدود (نا متناهی):

$$\mathrm{XOR}(x) = \sum_{i=0}^{|x|-1} x_i \quad , \quad x \in \{0,1\}^*$$

ما در این قسمت به سراغ توابعی به صورت $F:\{0,1\}^* \to \{0,1\}$ یا $F:\{0,1\}^* \to \{0,1\}^*$. این **به این معنی نیست که طول ورودی بی‌نهایت است، بلکه به این معنی است که طول ورودی می‌تواند متغیر باشد و هر مقداری باشد**. (بنابراین نمی‌توان آن را به صورت یک جدول که برای هر ورودی یک خروجی را نشان می‌دهد، نوشت)

یک تابع به شکل $F:\{0,1\}^* \to \{0,1\}^*$ یک تسک محاسباتی (computational task) را مشخص می‌کند ه یک مپ کردن از از ورودی‌هایی به شکل $x \in \{0,1\}^*$ به خروجی تابع $F(x)$ می‌باشد.

هر چیزی می‌تواند به صورت رشته باینری (binary string) ذخیره شود.

تابع TWIN PRIME به صورت زیر تعریف می‌شود: (می‌گوید آیا در اعداد بزرگ‌تر از x آیا دو عدد اول که با هم دو تا فاصله داشته باشند وجود دارد یا نه)

$$TWINP(x) = \begin{cases} 1 & \exists_{p\in\mathbb{N}} \quad p > |x| \quad \text{و} \quad p \text{ و } p+2 \text{ اول باشند} \\ 0 & \text{در غیر این صورت} \end{cases}$$

این تابع به صورت ریاضی تعریف شده‌است و برای هر ورودی، یا مقدار صفر یا یک را اختیار می‌کند. این تعریف یک specification است. همچنین **برنامه پایتونی که این را محاسبه کند وجود ندارد.**

6.1.1 – Varying inputs and outputs :

خیلی از توابع، بیشتر از یک (دو یا بیشتر) ورودی دارند. مثل تابع ضرب که به صورت مقابل است: $MULT(x,y) = x.y$. با توجه به اینکه ما می‌توانیم دوتایی (pair) از رشته‌های باینری را به صورت یک رشته باینری در نظر بگیریم، پس میتوانیم بگوییم تابع ضرب یک نگاشت (map) از $\{0,1\}^*$ به $\{0,1\}^*$ است. (یعنی: $MULT:\{0,1\}^* \to \{0,1\}^*$ است)

تابع پالیندورم (PALINDORM) به صورت زیر است:

$$PALINDORM(x) = \begin{cases} 1 & \forall_{i\in[|x|]}\ x_i = x_{|x|-i} \\ 0 & \text{در غیر این صورت} \end{cases}$$

اینگونه توابع که فقط یک بیت خروجی دارند، توابع بولینی (Boolean Function) نامیده می‌شوند. (از انواع توابع نامحدود (نامتناهی) هم به شمار می‌روند طبیعتا چون ورودی‌شان، رشته‌ای با طول نامحدود است)

بولینی کردن توابع (“Booleanizing” functions) یعنی تبدیل یک تابع به شکل تابع بولینی آن. مثلا یعنی برای تابع ضرب، تابع بولینی آن، تابعی است که خروجی، بیت i اُم را می‌دهد. یعنی:

$$MULT(x,y,i) = \begin{cases} i^{th}\ bit\ of\ x.y & i < |x.y| \\ 0 & \text{در غیر این صورت} \end{cases}$$

نحوه کلی بولینی کردن توابع:

اگر تابع $F:\{0,1\}^* \to \{0,1\}^*$ داشته باشیم، تابع بولینی آن به صورت زیر است:

$$BF(x,i,b) = \begin{cases} F(x)_i & i < |F(x)| \ \text{و}\ \ b = 0 \\ 1 & i < |F(x)| \ \text{و}\ \ b = 0 \\ 0 & i < |F(x)| \end{cases}$$

دلیل وجود b در این تابع چیست؟ دلیل آن فقز برای این است که در برنامه پایتون متناظر با این کار، بتواند طول ورودی را تشخیص دهد. (اگر B را برابر با یک بگذاریم، خروجی این تابع از اندیس صفر تا طول ورودی برابر با یک است و در بقیه موارد برابر با صفر است. بدین شکل می‌توان از این تابع برای محاسبه طول ورودی هم استفاده کرد)

Formal Languages – 6.1.2 :

برای هر تابع بولینی‌ای (Boolean Function) به صورت $F:\{0,1\}^* \to \{0,1\}$ ما می‌توانیم مجموعه‌ای از رشته‌ها (string) که خروجی تابع به ازای آن‌ها یک است را تعریف کنیم (تعریف ریاضی آن به صورت $L_F = \{x|\ F(x) = 1\}$ است) و به آن زبان (Language) می‌گوییم. یک زبان رسمی (formal language) یک زیرمجموعه از $L \subseteq \{0,1\}^*$ است. (اگر بخواهیم به صورت جامع‌تر بر روی الفبای (محدود) مورد استفاده (کاراکترهای مورد استفاده که تعدادشان محدود است) تعریف کنیم، می‌شود $L \subseteq \sum^*$)

حالا سوال عضویت (membership) یا تصمیم برای یک زبان، در اصل تسک مشخص‌کردن اینکه ورودی $x \in \{0,1\}^*$ متعلق به زبان L هست یا خیر می‌شود. ($x \in L$)

حالا اگر بتوانیم تابع F را محاسبه کنیم، می‌توانیم مسئله تصمیم عضویت (آیا ورودی متعلق به زبان پذیرش تایع هست یا نه) را حل کنیم و برعکس. (اگر بتونیم مسئله تصمیم عضویت یک ورودی به زبان پذیرش تابع را حل کنیم، می‌توانیم تابع را محاسبه کنیم)

Restriction of Function – 6.3.1 :

اگر تابع $F:\{0,1\}^* \to \{0,1\}$ یک تابع بولینی باشد و $n \in \mathbb{N}$ باشد، آنگاه محدودشده تابع F بر روی طول ورودی برابر با n ، که با F_n نشان‌داده می‌شود، یک تابع محدود است که به صورت $f:\{0,1\}^n \to \{0,1\}$ است به طوریکه برای هر رشته‌ای مانند x به طول n، مقدار توابع f و F با یکدیگر برابر باشد. (یعنی $F(x) = f(x)$ باشد به ازای تمامی $x \in \{0,1\}^n$)

Circuit collection for every infinite function – Theorem 6.1 :

اگر تابع $F:\{0,1\}^* \to \{0,1\}$ باشد، پس یک مجموعه (collection) از مدارها به صورت $\{C_n\}_{n\in\{1,2,\ldots\}}$ وجود دارد که به ازای هر $n > 0$، C_n محدود شده تابع F را بر روی ورودی‌هایی به طول n محاسبه می‌کند.

یعنی برای هر تابع نامحدود می‌توان توابع محدود شده با طول مشخص ورودی برایش آورد.

اثبات: این قضیه، یک نتیجه بدیهی از قضیه جامعیت مدارهای بولینی (Universality of Boolean circuit) است. (چون F_n یک نگاشت (mapping) از $\{0,1\}^n$ به $\{0,1\}$ است، پس یک تابع بولینی‌ای وجود دارد که آن را محاسبه کند. در حقیقت سایز آن مدار حداکثر $c.\frac{2^n}{n}$ است)

طبق این قضیه گفته شده، حتی برای تابع $TWINP$ هم یک همچین مجموعه C_n ای وجود دارد؛ حتی با وجود اینکه نمیدانیم آیا برای این تابع، برنامه‌ای (program) ای (به زبان پایتون یا ...) وجود دارد یا خیر. مشکل آن هم این است که برای هر (همه) مقدار n ای نمیدانیم که حلقه (circuit) موجود در C_n چیست. (برای یک مقدار می‌دانیم)

6.2 – DETERMINISTIC FINITE AUTOMATA:

با DFA ها از قبل آشنا هستیم. این DFA ها در توان محاسباتی برابر هستند با regular expressions. این *Regular expressions* ها یک مکانیزم قوی برای تشخیص الگو (pattern) هستند.

در سطح‌های بالا (high level)، الگوریتم یک دستور العمل (recipe) رسیدن به خروجی از روی یک ورودی با انجام ترکیبی از قدم‌های زیر است:

۱- خواندن یک بیت از ورودی ۲- بروزرسانی کردن حالت (state) کنونی ۳- تموم کردن و تولید خروجی

تعریف **single-pass constant-memory algorithm** :

یک بار ورودی را پیمایش می‌کنیم (single pass over the input) و همچنین حافظه‌ای (موری‌ای) که با آن کار می‌کند، محدود است. این الگوریتم به **DFA** یا همان Deterministic Finite Automata هم معروف است. (یک نام دیگر برای این *DFA* ها، *finite state machine* می‌باشد)

این الگوریتم‌ها را می‌توان مانند ماشینی که C تا وضعیت (state) دارند هم در نظر گرفت که در یک حالت شروعی قرار دارد و شروع به خواندن ورودی‌ها می‌کند و وضعیت (state) خود را بروزرسانی می‌کند. این بروزرسانی با توجه به وضعیت قبلی و ورودی انجام می‌شود) خروجی این ماشین نیز وابسته به وضعیتی است که در انتها در آن قرار می‌گیرد.

اگر الگوریتم c تا بیت از حافظه را استفاده کند، در آن صورت محتوای داخل آن در یک رشته به طول c قابل نمایش است. بنابراین این الگوریتم می‌تواند حداکثر در یکی **از 2^c تا وضعیت (state) مختلف** در هر مرحله از اجرا قرار بگیرد.

از این رو، ما می‌توانیم یک *DFA* که C تا وضعیت دارد را با لیستی از $C\,.\,2$ تا قاعده (قانون یا rule) مشخص کنیم که به آن **قاعده حرکت (transition rules)** هم گفته می‌شود. (هر قاعده به این صورت است که اگر در وضعیت (state) v بودیم و بیت ورودی σ بود، به وضعیت(state) جدید مثل v' برو.) و برای انتهای کار هم یک قاعده داریم که به این صورت است: اگر در وضعیت نهایی در یکی از وضعیت‌های ... بود (لیستی از وضعیت‌ها آورده می‌شود) آن وقت خروجی برابر با یک است در غیر این صورت خروجی برابر با صفر است.

ما می‌توانیم یک *DFA* با C تا وضعیت (state) را با یک گراف لیبل‌دار (Labeled Graph) هم توصیف کنیم. (برای هر وضعیت (state) یک خروجی به ازای هر عضو $\sum$ مثل σ قرار می‌دهیم. همچنین یک سری از گره‌ها را هم با عنوان گره پایانی قابل قبول (accepting states) مشخص می‌کنیم.)

Deterministic Finite Automata – Definition – 6.2:

> **Definition 6.2 — Deterministic Finite Automaton.** A deterministic finite automaton (DFA) with C states over $\{0,1\}$ is a pair $(T, \mathcal{S})$ with $T : [C] \times \{0,1\} \to [C]$ and $\mathcal{S} \subseteq [C]$. The *finite function* T is known as the *transition function* of the DFA. The *set* $\mathcal{S}$ is known as the set of *accepting states*.
>
> Let $F : \{0,1\}^* \to \{0,1\}$ be a Boolean function with the infinite domain $\{0,1\}^*$. We say that $(T, \mathcal{S})$ *computes* a function $F : \{0,1\}^* \to \{0,1\}$ if for every $n \in \mathbb{N}$ and $x \in \{0,1\}^n$, if we define $s_0 = 0$ and $s_{i+1} = T(s_i, x_i)$ for every $i \in [n]$, then
>
> $$s_n \in \mathcal{S} \Leftrightarrow F(x) = 1$$

توجه داشته باشید که تابع حرکت (**transition function**) یک تابع محدود است که با یک **جدول از قواعد** (**table of rules**) قابل نمایش است.

چندین روش برای تعریف DFA ها وجود دارد. مثلا می‌توان DFA را با یک پنج‌تایی $(Q, \sum, \delta, q_0, F)$ تعریف کرد. که در آن Q همان مجموعه وضعیت‌ها، $\sum$ همان الفبا (alphabet) است، δ همان تابع حرکت (transition function)، q_0 همان وضعیت شروع و F هم همان وضعیت‌های پذیرش (*accepting state*) است.

(هر وقت که از ما خواستند یک DFA بکشیم، کمک‌کننده‌ست که با کشیدن یک *single-pass constant-memory* شروع کنیم و الگوریتم آن را بدهیم و این کار را با روش‌های کلی‌تر مصل شبه‌کد یا کد پایتون یا … انجام دهیم.)

Figure – 6.4 : جدولی که برای کشیدن تابع حرکت (transition function) است، شامل سه ستون است که ستون اول وضعیت (state) کنونی، ستون دوم حرف الفبار ورودی و ستون سوم وضعیت جدید است.

Anatomy of an automaton (finite vs. unbounded) – 6.2.1 :

کمیت‌هایی با **سایز محدود** در DFA ها که به **سایز ورودی وابسته نیستند**:

- تعداد وضعیت‌ها (state)
- تابع حرکت (Transition function) که $2C$ تا ورودی دارد و می‌تواند با جدولی با تعداد سطر $2C$ مدل شود که خروجی هر کدام از این سطرها یک عدد در [C] است
- مجموعه S ⊆ [C] که شامل وضعیت‌های پذیرش (accepting states) است

پس فارغ از توضیحات داده شده در رابطه با الگوریتم‌ها، باید بتوان توصیف (description) یک DFA را به صورت کامل نوشت.

کمیت‌هایی با **سایز نامحدود** در DFA که **به هیچ عدد ثابتی محدود نمی‌شوند**:

- طول رشته ورودی
- تعداد مراحل (step) هایی که DFA طی می‌کند که می‌تواند با توجه به طول ورودی رشد کند. در اصل **با توجه به اینکه** ***single-pass*** **است، پس تعداد مراحل، دقیقا برابر با سایز ورودی** است.

6.2.2– DFA-Computable functions:

تابع $F:\{0,1\}^* \to \{0,1\}$، قابل محاسبه با *DFA* است اگر یک *DFA* وجود داشته باشد که تابع F را محاسبه کند.

Theorem 6.4 – DFA computable functions are countable:

اگر *DFACOMP* مجموعه تمام تابع‌های بولینی‌ای مانند $F:\{0,1\}^* \to \{0,1\}$ باشد به طوریکه یک *DFA* ای وجود داشته باشد که F را محاسبه کند. آن وقت *DFACOMP*، قابل شمارش (countable) است.

ایده اثبات: هر *DFA* را می‌توان با یک رشته با طول محدود (finite length string) نمایش داد. که منجیر به یک تناظر پوشا (onto map) بین $\{0,1\}^*$ به *DFACOMP* می‌شود. یعنی تابعی که متناظر می‌کند رشته این *DFA* را با تابعی که آن *DFA* آن را محاسبه می‌کند.

اثبات: هر *DFA* ای را می‌توان با یک رشته که شامل تابع حرکت (transition function) آن و استیت‌های پذیرش (accepting states) آن است، نمایش داد از طرفی هر *DFA* یک تابع $F:\{0,1\}^* \to \{0,1\}$ را محاسبه می‌کند. بنابراین ما تابع $StDC:\{0,1\}^* \to DFACOMP$ را به صورت زیر تعریف کنیم:

$$StDC(a) = \begin{cases} F & \text{تابعی که اتوماتا محاسبه می‌کند} \\ ONE & \text{در غیر این صورت} \end{cases}$$

تابع ONE تابع ثابت یک است که به صورت $ONE:\{0,1\}^* \to \{0,1\}$ تعریف می‌شود و خروجی آن همیشه یک است. حالا با توجه به اینکه هر تابع F ای داخل *DFACOMP*، با اتوماتایی قابل محاسبه است، پس *StDC* یک تابع پوشا (onto) است که دامنه آن $\{0,1\}^*$ و برد آن *DFACOMP* است. که یعنی *DFACOMP* قابل شمارش (Countable) است.

حالا با توجه به اینکه تمام توابع بولینی غیرقابل شمارش (Uncountable) هستند، پس می‌توانیم نتیجه بدیهی زیر را بگیریم:

Theorem 6.5 – Existence of DFA-uncomputable functions:

تابع بولینی وجود دارد که با هیچ *DFA* ای قابل محاسبه نیست.

اثبات: اگر تمام تابع‌های بولینی قابل محاسبه با *DFA* ها بودند، آن وقت *DFACOMP* مساوی مجموعه *ALL* (*All* of Boolean functions) میشد؛ اما این مجموعه ناشماراست (uncountable) است و با قضیه‌های قبلی در تضاد است.

6.3 – REGULAR EXPRESSIONS:

ما می‌توانیم به کاربر اجازه دهیم که الگو را با تعین کردن یک تابع قابل محاسبه ($F:\{0,1\}^* \to \{0,1\}$ computable function) مشخص کند؛ که این تابع *F(x) = 1* متناظر با پیدا کردن (matching) *x* باشد.

ما نمی‌خواهیم اجازه بدهیم که در **حلقه بی‌نهایت (infinite loop)** قرار بگیریم، برای همین به کاربر اجازه استفاده از ابزارهای کامل زبان برنامه‌نویسی را نمی‌دهند.

در *regular expression* ها دو عملیات | که برابر با چسباندن (concatenation) و * که نمایانگر تکرار است را داریم. بقیه موارد عموما با syntactic sugar ها از همینا بدست میان.

Regular Expression – Definition 6.6:

یک regular expression در الفبای (alphabet) $\sum \cup \{(,),|,*,\emptyset,""\}$ دارای یکی از فرم‌های زیر است:

- $e = \sigma$ اگر σ عضوی از $\sum$ باشد
- $e = (e'|e")$ در حالی که e' و e'' خودشان regular expression باشند
- $e = (e')\ (e")$ در حالی که e' و e'' خودشان regular expression باشند (عموما پرانتز را نمی‌گذاریم)
- $e = (e')^*$ به طوریکه e' خودش regular expressions باشدپ

در انتها ما اجاره وجود این حالت خاص و مهم $e = \emptyset$ و $e = ""$ را هم می‌دهیم. که اولی ($e = \emptyset$) مربوط به پذیرش هیچ رشته‌ای (no string) و دومی ($e = ""$) مرتبط با پذیرش رشته خالی (empty string) است.

بالاترین اولویت با * است، بعدش با چسباندن (concatenation) و بعدش با |. همچنین عملگر یا left associative است.

هر regular expression ای متناظر با یک تابع $\phi_e : \sum \rightarrow \{0,1\}$ است به طوریکه $\phi_e(x) = 1$ است اگر x مطابق (match) باشد با regular expression.

Matching a regular expression – Definition 6.7:

Definition 6.7 — Matching a regular expression. Let e be a regular expression over the alphabet Σ. The function $\Phi_e : \Sigma^* \rightarrow \{0,1\}$ is defined as follows:

1. If $e = \sigma$ then $\Phi_e(x) = 1$ iff $x = \sigma$.
2. If $e = (e'|e'')$ then $\Phi_e(x) = \Phi_{e'}(x) \vee \Phi_{e''}(x)$ where $\vee$ is the OR operator.
3. If $e = (e')(e'')$ then $\Phi_e(x) = 1$ iff there is some $x', x'' \in \Sigma^*$ such that x is the concatenation of x' and x'' and $\Phi_{e'}(x') = \Phi_{e''}(x'') = 1$.
4. If $e = (e')*$ then $\Phi_e(x) = 1$ iff there is some $k \in \mathbb{N}$ and some $x_0, \ldots, x_{k-1} \in \Sigma^*$ such that x is the concatenation $x_0 \cdots x_{k-1}$ and $\Phi_{e'}(x_i) = 1$ for every $i \in [k]$.
5. Finally, for the edge cases $\Phi_\emptyset$ is the constant zero function, and $\Phi_{""}$ is the function that only outputs 1 on the empty string "".

We say that a regular expression e over Σ *matches* a string $x \in \Sigma^*$ if $\Phi_e(x) = 1$.

یک تابع بولینی (Boolean function) **regular** نامیده می‌شود اگر خروجی‌اش دقیقا در مجموعه‌ای از رشته‌ها که با یک regular expression، match شوند، یک باشد.

Regular functions/languages – Definition 6.8:

اگر $\sum$ یک کجکوعه متناهی (محدود) باشد و $F: \sum^* \rightarrow \{0,1\}$ یک تابع بولینی باشد، میگوییم F یک تابع *regular* است اگر $F = \phi_e$ برای یک regular expression باشد. همچنین برای تمام زبان‌های رسمی (formal languages) مثل $L \subseteq \sum^*$ می‌گوییم L، *regular* است **اگر و تنها اگر** یک regular expression وجود داشته باشد به طوریکه $x \in L$ **اگر و تنها اگر** e با x، match باشد.

Algorithm for matching regular expression – 6.3.1:

Regular Expression matching – Algorithm 6.10:

Algorithm 6.10 — Regular expression matching.

Input: Regular expression e over Σ^*, $x \in \Sigma^*$
Output: $\Phi_e(x)$

procedure MATCH(e,x)
  if $e = \emptyset$ **then return** 0 **;**
  if $x =$ "" **then return** MATCHEMPTY(e) **;**
  if $e \in \Sigma$ **then return** 1 iff $x = e$ **;**
  if $e = (e'|e'')$ **then return** MATCH(e', x) or MATCH(e'', x) **;**
  if $e = (e')(e'')$ **then**
    for $i \in [|x|]$ **do**
      if MATCH$(e', x_0 \cdots x_{i-1})$ and MATCH$(e'', x_i \cdots x_{|x|-1}$ **then return** 1 **;**
    end for
  end if
  if $e = (e')^*$ **then**
    if $e' =$ "" **then return** MATCH$($"", $x)$ **;**
      # $($""$)^*$ *is the same as* ""
    for $i \in [|x|]$ **do**
      # $x_0 \cdots x_{i-1}$ *is shorter than* x
      if MATCH$(e, x_0 \cdots x_{i-1})$ and MATCH$(e', x_i \cdots x_{|x|-1})$ **then return** 1 **;**
    end for
  end if
  return 0
end procedure

نکته کلیدی این است که در تعریف بازگشتی (recursive) regular expression ها، هر وقت e از دو تا regular expression مثل e' و e'' تشکیل شده بود، آن وقت e' و e''، از e **کوچک‌تر** هستند. و وقتی که سایز آن‌ها به یک برسد، باید مطابق با یکی از حالت‌های غیربازگشتی باشند. بنابراین با توجه به این نکته، فراخوانی‌های بازگشتی الگوریتم بالا همیشه منجر به **کوچک‌تر شدن *regular expression* یا در حالت $(e')^*$ منجر به کوچک‌تر شدن طول ورودی** می‌شود.

Matching Empty String – Solved Exercise 6.3 :

می‌توانیم برای این کار یک الگوریتم بازگشتی با مشاهدات زیر پیدا کنیم:

۱. عبارتی به شکل "" یا $(e')^*$ همواره رشته خالی را پذیرش (match) می‌کند.
۲. عبارتی به فرم σ در حالیکه $\sigma \in \sum$ باشد (در الفبای ما باشد)، هیچگاه رشته خالی (match) را پذیرش نمی‌کند.
۳. Regular expression ای با فرم $\emptyset$ رشته خالی را پذیرش (match) نمی‌کند.
۴. عبارتی به فرم $e'|e''$، رشته خالی را پذیرش می‌کند (match) اگر و تنها اگر e' یا e'' رشته خالی را پذیرش کنند.
۵. عبارتی به فرم $(e')(e'')$ رشته خالی را پذیرش می‌کند اگر و تنها اگر هر دو عبارت e' و e'' رشته خالی را پذیرش کنند.

Algorithm 6.11 — Check for empty string.

Input: Regular expression e over Σ^*, $x \in \Sigma^*$
Output: 1 iff e matches the emptry string.

procedure MATCHEMPTY(e)
  if $e =$ "" **then return** 1 **;**
  if $e = \emptyset$ or $e \in \Sigma$ **then return** 0 **;**
  if $e = (e'|e'')$ **then return** MATCHEMPTY(e') or MATCHEMPTY(e'') **;**
  if $e = (e')(e'')$ **then return** MATCHEMPTY(e') and MATCHEMPTY(e'') **;**
  if $e = (e')^*$ **then return** 1 **;**
end procedure