بسمه تعالی

دانشکده مهندسی برق و کامپیوتر

دانشگاه صنعتی اصفهان

طراحی کامپایلرها – نیمسال دوم ۹۹-۱۳۹۸

تکلیف شماره دو

مریم سعید مهر

ش.د. : ۹۶۲۹۳۷۳

۱- در مورد گرامر و رشته‌ی ورودی زیر به سوالاتی که در ادامه آمده است پاسخ دهید :

S → S + S | S S | (S) | S* | a with string (a + a) * a

الف) یک انشقاق چپ (left-most derivation) برای رشته‌ی ورودی بنویسید.

ب) یک انشقاق راست (right-most derivation) برای رشته‌ی ورودی بنویسید.

ج) یک درخت تجزیه برای این رشته بنویسید.

د) آیا این گرامر مبهم است یا غیرمبهم ؟ توضیح دهید.

ه) زبانی که این گرامر تولید میکند را شرح دهید.

پاسخ :

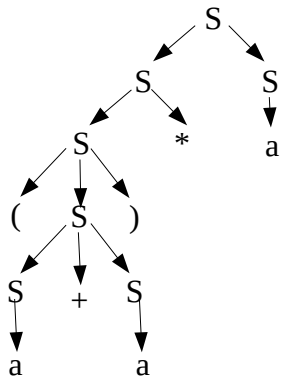
الف)

$S \rightarrow S\,S \rightarrow S * S \rightarrow (\,S\,) * S \rightarrow (\,S + S\,) * S \rightarrow (\,a + S\,) * S \rightarrow (\,a + a\,) * S \rightarrow (\,a + a\,) * a$

ب)

$S \rightarrow S\,S \rightarrow S\,a \rightarrow S * a \rightarrow (\,S\,) * a \rightarrow (\,S + S\,) * a \rightarrow (\,S + a\,) * a \rightarrow (\,a + a\,) * a$

ج)



د) این گرامر مبهم است ، همانطور که در قسمت (الف) و (ب) دیدیم ، دو انشقاق مختلف برای یک عبارت ورودی وجود داشت که این نشانگر ابهام گرامر است.
جهت رفع ابهام ، میبایست اولویت گذاری کنیم . (مشابه گرامر عبارات ریاضی!)

ه) این گرامر سعی دارد به نوعی RE ها را تولید کند ، به این شکل که + بیانگر or است و عدم وجود هیچ اوپراتوری (منظور قانون SS است) به معنای concat می‌باشد و * همان Kleene Star است .

۲- گرامر را به گرامری تبدیل کنید که بازگشتی چپ مستقیم یا ضمنی نداشته باشد.

A → Ba | Aa | c

B → Bb | Ab | d

پاسخ :

ابتدا بازگشتی های مستقیم B را حذف میکنم :

$B \rightarrow AbB' \mid dB'$

$B' \rightarrow bB' \mid \varepsilon$

حالا با درنظر گرفتن قانون جدید B ، آنها را در قانون A جایگزین میکنم :

$A \rightarrow AbB'a \mid dB'a \mid Aa \mid c$

حالا بازگشتی های مستقیم A را نیز حذف میکنم :

$A \rightarrow dB'aA' \mid cA'$

$A' \rightarrow bB'aA' \mid aA' \mid \varepsilon$

در نهایت ، گرامر اصلاح شده به صورت زیر است :

$A \rightarrow dB'aA' \mid cA'$

$A' \rightarrow bB'aA' \mid aA' \mid \varepsilon$

$B \rightarrow AbB' \mid dB'$

$B' \rightarrow bB' \mid \varepsilon$

۳- گرامر زیر تلاشی جهت نوشتن گرامر عبارات شرطی بدون ابهام است . آیا ابهام برطرف شده است ؟ اگر هنوز گرامر مبهم است ، نشان دهید که ابهام وجود دارد ؟

$$
\begin{array}{lcl}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & \mid & matchedStmt \\
matchedStmt & \rightarrow & \textbf{if } expr \textbf{ then } matchedStmt \textbf{ else } stmt \\
 & \mid & \textbf{other}
\end{array}
$$

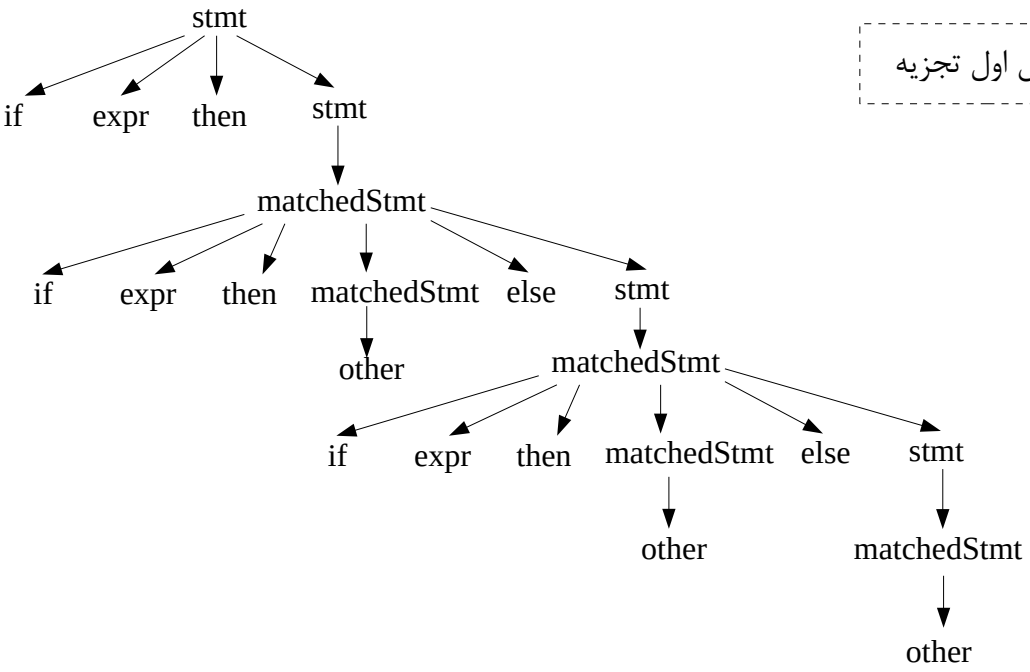
پاسخ :

این گرامر همچنان مبهم است ، زیرا :

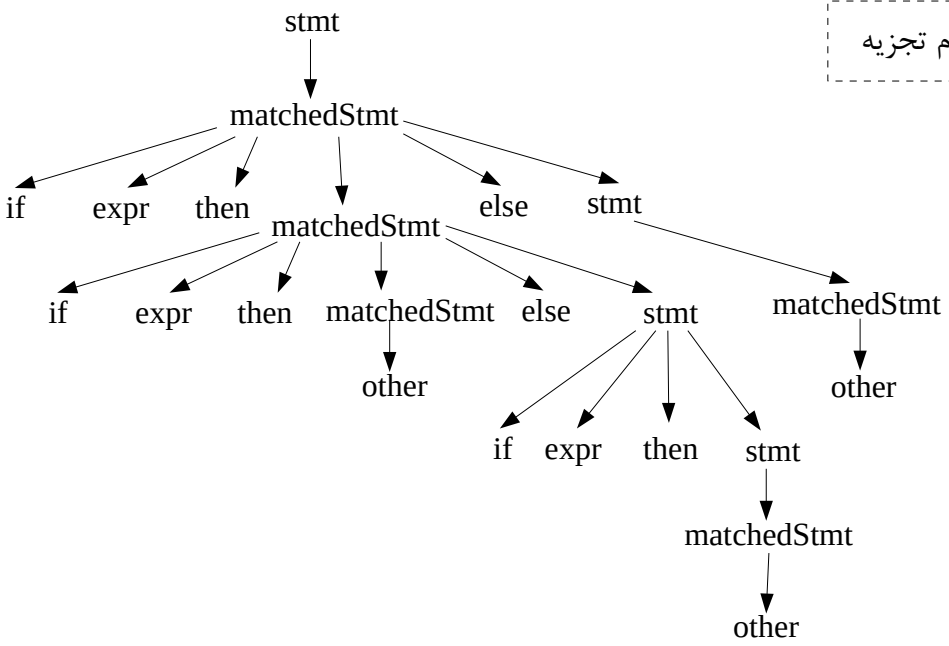
if if else if else

عبارت فوق را به دو شیوه میتوان تجزیه کرد ، به همین سبب گرامر فوق دارای ابهام است .

روش اول تجزیه



روش دوم تجزیه



۴- برای گرامر زیر یک sudocode برای اجرای الگوریتم Recursive Descent به صورت بازگشتی بنویسید .

$$
\begin{aligned}
S &\rightarrow S \text{ and } S \\
&\mid S \text{ or } S \\
&\mid T \\
&\mid \text{a} \\
T &\rightarrow \text{a}
\end{aligned}
$$

پاسخ :

اگر بخواهیم برای این گرامر که ابهام ، left recursion دارد و همچنین می‌توان فاکتورگیری از چپ کرد ، الگوریتم RD را اجرا کنیم ، دچار چندگانگی درخت پارس نهایی(به دلیل ابهام گرامر) و لوپ می‌شود اما به هر حال سودوکد آن را می‌نویسم و همچنین در ادامه همین سوال را با اجرای الگوریتم predictive recursive حل میکنم.

```
bool S() {
  Token *saved = next;
  return ( S1() || (next=saved , S2()) || (next=saved , T()) || (next=saved , match('a')));
}

bool S1() { return ( S() && match('and') && S() ); }

bool S2() { return ( S() && match('or') && S() ); }

bool T() { return match('a') ; }

bool match(Token tok) { return *next++ == tok ; }
```

گرامر فوق هم دارای ابهام است هم left recursion ، اول دو اشکال گفته شده را با اولویت گذاری و سپس حذف left recursion برطرف کرده و سپس سودوکد خواسته شده را مینویسم

$$
\begin{aligned}
&S \rightarrow S \text{ or } R \mid R \\
&R \rightarrow R \text{ and } T \mid T \\
&T \rightarrow \text{a}
\end{aligned}
\quad \xrightarrow{\text{Left recursion}} \quad
\begin{aligned}
&S \rightarrow RS' \\
&S' \rightarrow \text{or } RS' \mid \varepsilon \\
&R \rightarrow TR' \\
&R' \rightarrow \text{and} TR' \mid \varepsilon \\
&T \rightarrow \text{a}
\end{aligned}
$$

حالا الگوریتم RD را روی این گرامر پیاده می‌کنم :

```
bool S() {
  Token *saved = next;
  return (R() && S'());
}

bool S'() { return ((match('or') && R() && S'()) || epsilon()); }

bool R() { return (T() && R'()); }

bool R'() { return ((match('and') && T() && R'()) || epsilon()); }

bool T() { return match('a'); }

bool match(Token tok) { return *next++ == tok ; }

bool epsilon() { return TRUE; }
```

در ادامه پیاده سازی الگوریتم predictive recursive را روی همین گرامر اصلاح شده ، آورده ام :

```
S() {
   if (*next first(RS'))
   {
     R();
     S'();
   }
   else
     Syntax Error;
}
S'() {
   if (*next first(orRS'))
   {
     match('or');
     R();
     S'();
   }
   else if (*next follow(S'))
     return ;
   else
     Syntax Error;
}
```

```
R() {
  if (*next first(TR'))
  {
    T();
    R'();
  }
  else
    Syntax Error;
}
R'() {
  if (*next first(andTR'))
  {
    match('and');
    T();
    R'();
  }
  else if (*next follow(R'))
    return ;
  else
    Syntax Error;
}
T() {
  if (*next first(a))
    match('a');
  else
    Syntax Error;
}
match(Token tok) { return *next++ == tok; }
int main() {
  *next = next terminal at input and initialized by first terminal at input ;
  if (S())
     print("input is parsable") ;
  else
     Error() ;
  return 0 ;
}
```

۵- الف) بررسی کنید آیا می‌توان برای گرامر زیر یک پارسر LL1 طراحی کرد ؟ چرا ؟

$A \rightarrow Ax \mid aA \mid Ay \mid aab$

ب) تحقیق و بررسی کنید شروط لازم و کافی برای LLk بودن یک گرامر چیست و آن‌ها را بیان کنید سپس بررسی کنید آیا برای گرامر فوق می‌توان پارسر LLk طراحی کرد یا نه ؟ اگر پاسخ شما بله است برای کدام k ها ؟

ج) در صورتی که پاسخ شما به قسمت (الف) خیر است ، تلاش کنید برای گرامر فوق یک گرامر معادل طراحی کنید که LL1 باشد

د) با توجه به گرامر فوق بررسی کنید آیا می‌توان برای هر گرامر LLk یک گرامر معادل LL1 طراحی کرد ؟ علت پاسخ خود را توضیح دهید.

پاسخ :

(الف) اولاً در صورتی می‌توان برای یک گرامر ، پارسر LL(1) طراحی کرد که گرامر ، LL(1) باشد .

دوماً یکی از شروط لازم و کافی برای LL(1) بودن یک گرامر این است که :

if $A \rightarrow \alpha \mid \beta$ then : $first(\alpha) \cap first(\beta) = \emptyset$

بدیهی است که production دوم و چهارم قانون A ، دارای اشتراک مجموعه‌ی first هستند .

درنتیجه گرامر LL(1) نیست و نتیجتاً نمی‌توان برای آن پارسر LL(1) طراحی کرد.

(ب) بر اساس این لینک ، یک گرامر وقتی LL(K) است اگر برای هر قانون $A \rightarrow \alpha \mid \beta$ داشته باشیم

$$[\ first_k(\alpha) \oplus follow_k(A)\ ] \cap [\ first_k(\beta) \oplus follow_k(A)\ ] = \emptyset$$

توضیح نوتیشن های جدید :

For any terminal string α we write :

$first_k(\alpha)$ is the prefix of a of length k (or all of a if its length is less than k)

For any string γ of terminal and non-terminal symbols and any non-terminal symbol A,

we say $A \rightarrow^* \gamma$ if we can derive γ from A. In English we say A derives γ.

For any non-terminal symbol

$first_k(A)$ is the set of $first_k(\alpha)$ of all terminal strings A derives.

Note that for recursive descent to work, if $A \rightarrow B1 \mid B2$ is a grammar rule we need $first_k(B1)$ disjoint from $first_k(B2)$.

For any two sets S1 and S2 of strings of terminal symbols $S1 \oplus_k S2$ is the set of prefixes of length k of all of the strings you an get by concatenating a string from S1 with a string from S2.

For any non-terminal symbol A, $follow_k(A)$ is the set of prefixes of length k of all the terminal strings that could come after strings generated from symbol A. To be precise :

$$follow_k(A) = \{first_k(x) \mid S \rightarrow^* wAx \text{ any strings w and x}\}$$

با کمی بازی با شرط کافی و لازم گفته شده برای LL(K) بودن گرامر ، می‌توان دریافت که ، در اصل همان دو شرط گفته شده برای LL(1) بودن گرامر ، با زیروند k (و البته معانی جدیدی که این زیروند به مجموعه های first و follow می‌دهد) است ، یعنی برای هر قانون $A \rightarrow \alpha \mid \beta$ داریم :

1. $first_k(\alpha) \cap first_k(\beta) = \emptyset$
2. if $\beta \rightarrow^* \varepsilon$ then : $follow(A) \cap first(\alpha) = \emptyset$

حالا بررسی LL(K) بودن گرامر فوق :

(از آن جایی که هیچ قانونی برایمان ε تولید نمی‌کند لذا نیازی به محاسبه ی follow ها نداریم و صرفاً first ها را به ازای kهای مختلف محاسبه میکنم.)

در اصل ۴ قانون داریم :

$A \rightarrow aA \rightarrow first_1(aA)=\{a\}/first_2(aA)=\{aa\}/first_3(aA)=\{aaa\}/first_4(aA)=\{aaaa,aaab\}$
$A \rightarrow aab \rightarrow first_1(aab)=\{a\}/first_2(aab)=\{aa\}/first_3(aab)=\{aab\}/first_4(aab)=\{aab\}$

$\Longrightarrow first_1(A)=\{a\}/first_2(A)=\{aa\}/first_3(A)=\{aaa,aab\}/first_4(A)=\{aaaa,aabx,aaby,aaab\}$

$A \rightarrow Ax \rightarrow first_1(Ax)=\{a\}/first_2(Ax)=\{aa\}/first_3(Ax)=\{aab,aaa\}/first_4(Ax)=\{aabx\}$
$A \rightarrow Ay \rightarrow first_1(Ay)=\{a\}/first_2(Ay)=\{aa\}/first_3(Ay)=\{aab\}/first_4(Ay)=\{aaby\}$

از محاسبات فوق پیداست که ، تنها در k=4 برای اولین بار هر چهار production از قانون A کاملاً متمایز از یکدیگر هستند ، لذا این گرامر LL(4) است .

به علاوه ، کاملاً بدیهی است که :

$$LL(K) \subset LL(K+1)$$

پس برای گرامر فوق می‌توان یک پارسر LL(4) , LL(5) , LL(6).....LL(K) (برای هر $K \geq 4$) طراحی کرد.

(ج) گرامر فوق فاقد ابهام است ولی دارای left recursion است همچنین قابلیت فاکتورگیری از چپ دارد.

ابتدا مشکلات فوق را به امید LL(1) شدن حل می‌کنم :

رفع left recursion :

$A \rightarrow aAA' \mid aabA'$
$A' \rightarrow yA' \mid xA' \mid \varepsilon$

فاکتورگیری از چپ :

$A \rightarrow aB$
$A' \rightarrow yA' \mid xA' \mid \varepsilon$
$B \rightarrow AA' \mid abA'$

گرامر فوق به ظاهر دیگر ایرادی ندارد اما با بررسی first تمام production های قانون B داریم :

$first(AA') = first(A) = \{a\}$, $first(abA') = \{a\}$ => grammer is not LL(1)

پس گرامر باز هم LL(1) نیست . می‌شود یک بار دیگر از قانون B فاکتورگیری کرد ، یعنی :

$$\begin{array}{l} A \rightarrow aB \\ A' \rightarrow yA' \mid xA' \mid \varepsilon \\ B \rightarrow AA' \mid abA' \end{array} \xrightarrow{AA' \equiv aBA'} \begin{array}{l} A \rightarrow aB \\ A' \rightarrow yA' \mid xA' \mid \varepsilon \\ B \rightarrow aBA' \mid abA' \end{array}$$

حالا فاکتورگیری می‌کنیم:

$$\begin{array}{l} A \rightarrow aB \\ A' \rightarrow yA' \mid xA' \mid \varepsilon \\ B \rightarrow aC \\ C \rightarrow BA' \mid bA' \end{array}$$

حالا این گرامر کاملا درست است و هر دو شرط لازم و کافی LL(1) بودن را ارضا می‌کند .

(د) همان طور که در قسمت (ب) اشاره کردم ،

$$LL(K) \subset LL(K+1)$$

ولی عکس آن مطلقا امکان پذیر نیست چون در آن صورت دیگر به آن گرامر LL(K) نمی‌گفتند . در واقع برای هر گرامر که از نوع LL(K) باشد ، آن را با کمترین K ممکن معرفی می‌کنند زیرا برای بقیه ی K ها طبق همان رابطه فوق ، قابل تصمیم گیری است.

پس اگر گرامری LL(K) بود (با K>1) قطعا نمی‌تواند همزمان LL(1) هم باشد ولی LL(K+1) هست!

در مورد گرامر مورد بحث این سوال ، گرامر اولیه از نوع LL(4) بود ولی با تغییراتی ، آن را به یک گرامر جدید تبدیل کردیم که از قضا LL(1) است .

نتیجه این که اگر یک گرامر LL(K) باشد ، با حذف LR ها و فاکتورگیری از چپ تا حد ممکن، می‌توان آن را به فرم LL(1) تبدیل کرد.

۶- (سوال اختیاری) گرامر زیر را در نظر بگیرید و فرض کنید e , s ترمینال هستند. جهت حل ابهامی که در بسط اختیاری else (در غیرپایانی stmtTail) پیش می‌آید ، هر جا از ورودی ، else دیدیم آن را مصرف می‌کنیم. بدین ترتیب می‌توانیم یک تجزیه‌گر پیشگو (predictive parser) بسازیم. با استفاده از ایده سمبل‌های هماهنگ ساز (synchronizing) به سوالات زیر پاسخ دهید.

الف) برای این گرامر یک جدول تجزیه بسازید که خطاها را نیز اصلاح کند.

stmt	→	**if** *e* **then** *stmt stmtTail*
	|	**while** *e* **do** *stmt*
	|	**begin** *list* **end**
	|	*s*
stmtTail	→	**else** *stmt*
	|	ϵ
list	→	*stmt listTail*
listTail	→	; *list*
	|	ϵ

ب) عملیات تجزیه ی خود را برای دو نمونه ورودی دلخواه تست کنید که در یک ورودی خطایی باشد که با follow سمپل روی استک کنترل شود و در دیگری با first ساختار بالاتر .

پاسخ :

(الف)

در ابتدا محاسبه‌ی مجموعه های first , follow :

first (if e then stmt stmtTail) = {if e then}
first (while e do stmt) = {while e do}
first (begin list end) = {begin}
first (s) = {s}

first (else stmt) = {else}

first (stmt listTail) = first (stmt) = {if e then , while e do , begin , s}

first (; list) = { ; }

follow (stmt) = {$, else }
follow (stmtTail) = {$}
follow (list) = {end}
follow (listTail) = {end}

منظور از *Sync همان first ساختار بالاتر است که در مورد stmtTail همان stmt ساختار بالاتر محسوب میشود و در مورد listTail منظور list است. برای توضیح بیشتر

	if e then	while e do	begin	end	else	s	;	$
stmt	if e then stmt stmtTail	while e do stmt	begin list end		Sync	s		Sync
stmtTail	Sync*	Sync*	Sync*		else stmt	Sync*		ε
list	stmt listTail	stmt listTail	stmt listTail	Sync		stmt listTail		
listTail	Sync*	Sync*	Sync*	ε		Sync*	; list	

(ب)

First Example : while e do begin s; if e then else s end

Stack	Input	Action
stmt $	while e do begin s; if e then else s end $	stmt → while e do stmt
while e do stmt $	while e do begin s; if e then else s end $	Match(while e do)

stmt $	begin s; if e then else s end $	stmt → begin list end
begin list end $	begin s; if e then else s end $	Match(begin)
list end $	s; if e then else s end $	list → stmt listTail
stmt listTail end $	s; if e then else s end $	stmt → s
s listTail end $	s; if e then else s end $	Match(s)
listTail end $	; if e then else s end $	listTail → ; list
; list end $	; if e then else s end $	Match(;)
list end $	if e then else s end $	list → stmt listTail
stmt listTail end $	if e then else s end $	stmt → if e then stmt stmtTail
if e then stmt stmtTail listTail end $	if e then else s end $	Match(if e then)
stmt stmtTail listTail end $	else s end $	Error() , pop(stmt)
stmtTail listTail end $	else s end $	stmtTail → else stmt
else stmt listTail end $	else s end $	Match(else)
stmt listTail end $	s end $	stmt → s
s listTail end $	s end $	Match(s)
listTail end $	end $	listTail → ε
end $	end $	Match(end)
$	$	Match($)

در نهایت رشته‌ی فرض شده :

while e do begin s; if e then ✕ else s end

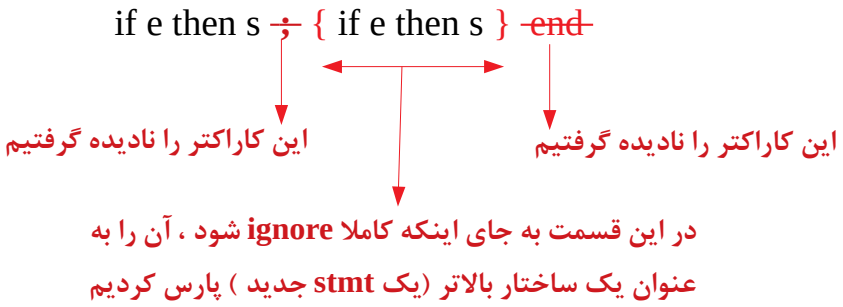
(فرض شد یک عبارت مثل s دیده شده)

Second Example : if e then s ; if e then s end

Stack	Input	Action
stmt $	if e then s ; if e then s end $	stmt → if e then stmt stmtTail
if e then stmt stmtTail $	if e then s ; if e then s end $	Match(if e then)
stmt stmtTail $	s ; if e then s end $	stmt → s
s stmtTail $	s ; if e then s end $	Match(s)
stmtTail $	; if e then s end $	Error() , skip(;)
stmtTail $	if e then s end $	Error() , Sync*=first(stmt)

if e then stmt stmtTail $	if e then s end $	Match(if e then)
stmt stmtTail $	s end $	stmt → s
s stmtTail $	s end $	Match(s)
stmtTail $	end $	Error() , skip(end)
stmtTail $	$	stmtTail → ε
$	$	Match($)

در نهایت رشته‌ی فرض شده :



مریم سعیدمهر-۹۶۲۹۳۷۳

تاریخ تحویل : ۱۳۹۹/۲/۲۷