# جواب تمرین سری اول درس کامپایلر

تهیه کنندگان:

زهرا اخلاقی

علیرضا صالحی حسین آبادی

استاد درس: زینب زالی

# ۱

- بیش از حد شدن طول شناسه‌ها یا ثابت‌های عددی( Exceeding length of identifier or numeric constants) – int a=2147483647 +1;
- ظاهرشدن حرف‌های غیرمجاز( Appearance of illegal characters) – printf(“Compiler");$
- عدم تطبیق رشته(Unmatched string) –

```
/* comment
cout<<"GFG!";
return 0;
```

- غلط املایی(Spelling Error) – int 3num= 1234

- جایگزینی یک حرف با یک حرف نادرست( Replacing a character with an incorrect character.) – int x = 12#34
- **حذف حرفی که باید حضور داشته باشد.**( Removal of the character that should be present.) – #include <istream>
- **جابجایی دو حرف**( Transposition of two characters.) – int mian()

# ۱(ادامه)

- بازیابی حالت پنیک(Panic mode recovery): در این روش، حرف‌های متوالی از ورودی یکی یکی حذف می‌شوند تا زمانی که مجموعه مشخصی از نشانه‌های همگام‌سازی پیدا شود. توکن های همگام سازی جداکننده هایی مانند؛ یا }
- جابجایی دو حرف مجاور.(Transpose of two adjacent characters)
- یک حرف از حذف شده را در ورودی باقیمانده وارد کنید.( Insert a missing character into the remaining input.)
- یک حرف را با حرف دیگری جایگزین کنید.(Replace a character with another character)
- یک حرف را از ورودی باقیمانده حذف کنید.(Delete one character from the remaining input.)

۲

| متغیر مورد نظر | تک گذره(onepass) | چندگذره(mutipass) |
|---|---|---|
| سرعت | سریع(fast) | آرام(slow) |
| حافظه | بیشتر(more) | کمتر(less) |
| زمان | کمتر(less) | بیشتر(more) |
| سادگی | خیر(no) | بله(yes) |

# ۳

- کد میانی کدی است که یک مرحله قبل از تولید کد ماشین تولید می شود و مستقل از ماشین هدف است و توانایی اجرا بر روی ماشین به طور مستقیم را ندارد، در حالی که کد نهایی کدی است که به زبان ماشین است و توسط ماشینی با سخت افزار مشخص قابل اجرا است پس وابسته به ماشین مورد نظر است. کد ماشین کدی است که تولید آن راحت‌تر از کد ماشین است و تبدیل آن به کد ماشین هدف هم راحت است. در کد میانی ترتیب اجرای اپراتورها مشخص شده است و به همان صورت است که در کد ماششین خواهد آمد اما مقداردهی رجیسترها هنوز صورت نگرفته و از متغیرهایی به جای رجیسترها استفاده می شود.
- کد میانی توسط قسمت front-end یک کامپایلر و از روی برنامه اصلی(source program) ساخته می‌شود و قسمت back-end از این کد میانی برای تولید کد نهایی(target code) استفاده می‌کند.
- به دلیل کد میانی مستقل از ماشین، قابلیت portability وجود دارد. برای مثال، فرض کنید، اگر یک کامپایلر بدون داشتن گزینه‌ای برای تولید کد میانی، زبان مبدأ را به زبان ماشین مقصد خود ترجمه کند، سپس برای هر ماشین جدید، یک کامپایلر بومی کامل ضروری است. زیرا بدیهی است که تغییراتی در خود کامپایلر با توجه به مشخصات دستگاه انجام شده است.
- هدف گذاری مجدد تسهیل می شود.
- اعمال اصلاح کد اصلی(source program) برای بهبود عملکرد کد اصلی(source program) با بهینه سازی کد میانی آسان‌تر است.

۴

| نوع | توکن | نوع | توکن | نوع | توکن |
|---|---|---|---|---|---|
| Symbol(Comma) | , | Symbol(Comma) | , | keyword | int |
| id | a | keyword | int | id | main |
| Operator(Addition) | + | id | b | Symbol(OLP) | ( |
| id | b | Operator(Assignment) | = | Symbol(CRP) | ) |
| Symbol(CRP) | ) | Constant | 10 | Symbol(OCB) | { |
| Symbol(Semi-Colon) | ; | Symbol(Semi-Colon) | ; | keyword | int |
| keyword | printf | keyword | printf | id | a |
| Symbol(OLP) | ( | Symbol(OLP) | ( | Operator(Assignment) | = |
| string | "i = %d, &i = %x" | string | "sum is :%d" | Constant | 10 |

# ۴(ادامه)

| توکن | نوع |
|---|---|
| , | Symbol(Comma) |
| i | id |
| , | Symbol(Comma) |
| & | Operator(Ampersand) |
| i | id |
| ) | Symbol(CRP) |
| ; | Symbol(Semi-Colon) |
| return | keyword |
| 0 | Constant |

| توکن | نوع |
|---|---|
| ; | Symbol(Semi-Colon) |
| } | Symbol(CCB) |

| | |
|---|---|
| ORP | Opening Left Parenthesis |
| CLP | Closing Right Parenthesis |
| OCB | Opening Curly Bracket |
| CCB | Closing Curly Bracket |

# ۵

- الف:

$$([0-9]^3) - [0-9]^3 - [0-9]^4$$

- ب:

$$\text{names} = \text{letters}^+$$

$$\text{names@names.names.names.names}$$

- ج:

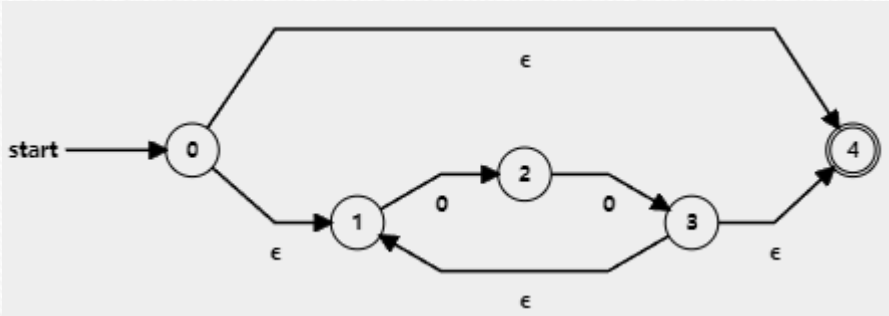
$$[-,+][0-9].[0-9]^+(e,E)[-,+][0-9]^+ \mid [0-9].[0-9]^+(e,E)[-,+][0-9]^+$$

- د:

$$[\text{' '}, \text{'\t'}, \text{'\n'}]^+$$

٦

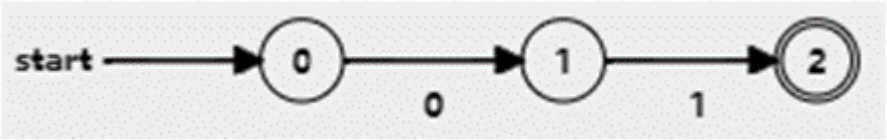
$(00)^*$




00

# ۶(ادامه)

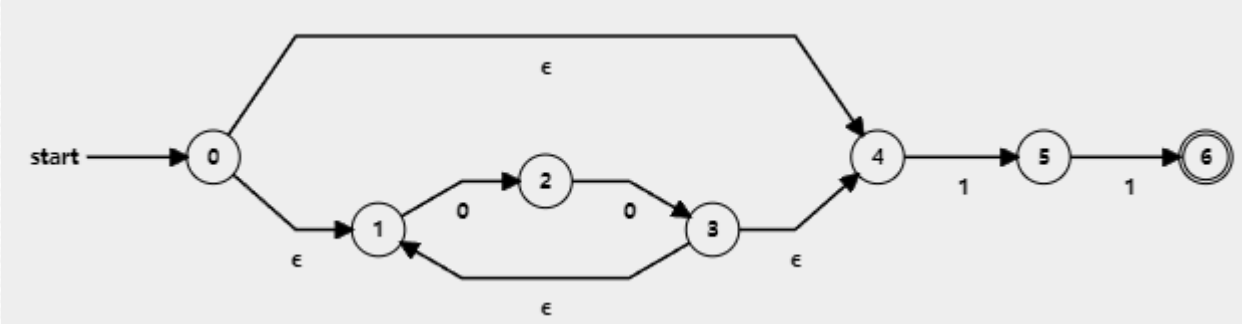
**01**




**11**

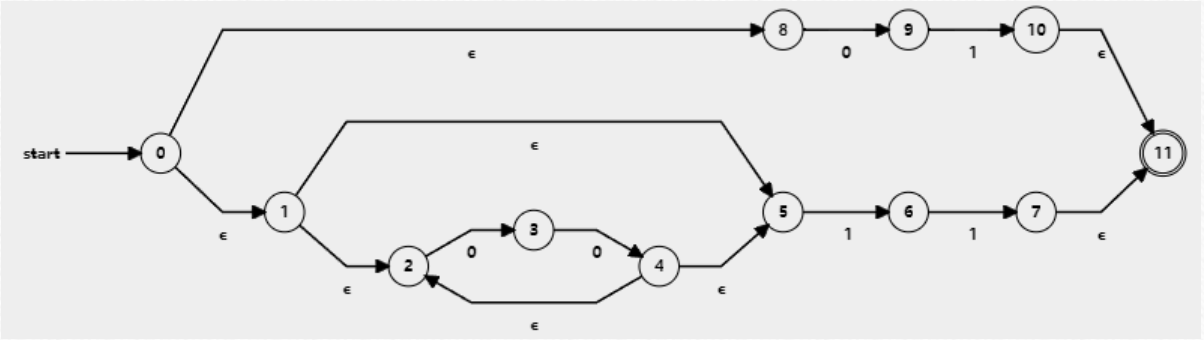
# ۶(ادامه)

$(00)^*11$



# ۶(ادامه)

$$(00)^*11 + 01$$

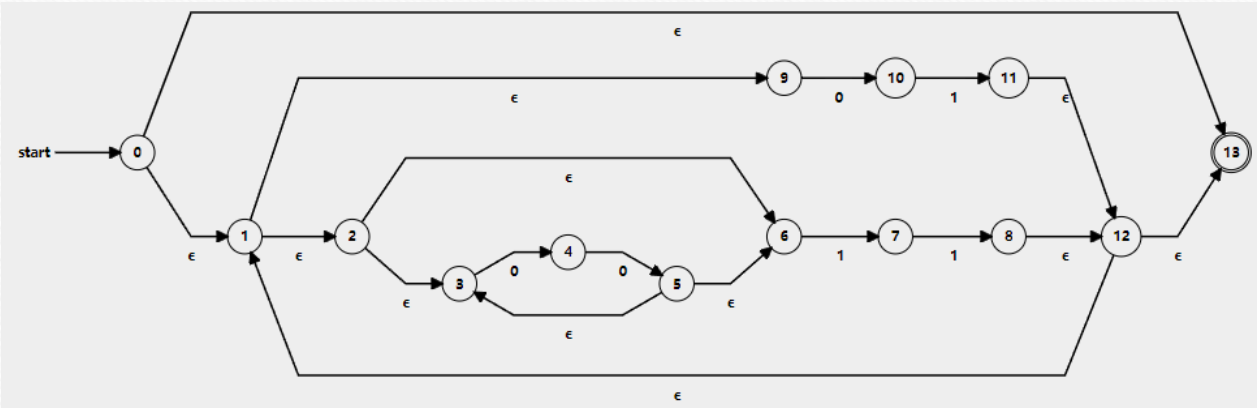


# ۶(ادامه)

$$(((00)^*11)+01)^*$$

$$\equiv \left(((00)^*11)^*(01)^*\right)^*$$

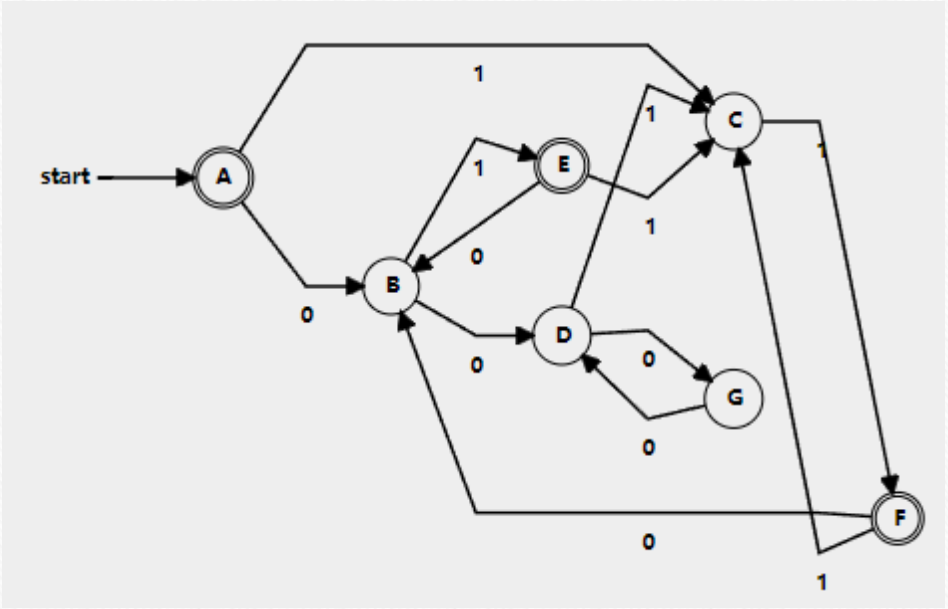
$$(X+Y)^* = (X^*Y^*)^*$$



# ۶(ادامه)

| NFA State | DFA State | Type | 0 | 1 |
|---|---|---|---|---|
| {0,1,2,3,6,9,13} | A | Accept | B | C |
| {4,10} | B | | D | E |
| {7} | C | | | F |
| {3,5,6} | D | | G | C |
| {1,2,3,6,9,11,12,13} | E | Accept | B | C |
| {1,2,3,6,8,9,12,13} | F | Accept | B | C |
| {4} | G | | D | |

# ۶(ادامه)



# ۶(ادامه)

- تقسیم بندی زیر را در نظر بگیرید:

$$P_0 = \{\{A, E, F\}, \{B, C, D, G\}\}$$

- با توجه به تابع Transition مربوط به DFA در اسلاید قبل داریم:
  - $\delta(B, 1) = E$
  - $\delta(D, 1) = C$

- بنابراین B و D قابل جدا شدن هستند، یعنی تقسیم بندی به صورت زیر در می‌آید:

$$P_1 = \{\{A, E, F\}, \{B, C, G\}, \{D\}\}$$

  - $\delta(B, 1) = E$
  - $\delta(G, 1) = -$
  - $\delta(B, 0) = D$
  - $\delta(C, 0) = -$

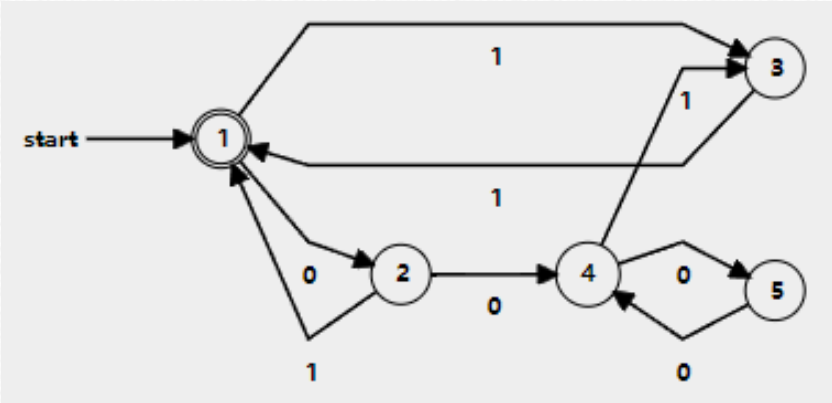
# ۶(ادامه)

| DFA State | Min-DFA State | TYPE | 0 | 1 |
|---|---|---|---|---|
| $\{A, E, F\}$ | 1 | Accept | 2 | 3 |
| $\{B\}$ | 2 | | 4 | 1 |
| $\{C\}$ | 3 | | | 1 |
| $\{D\}$ | 4 | | 5 | 3 |
| $\{G\}$ | 5 | | 4 | |

- بنابراین B و C و D نیز قابل جدا شدن هستند و تقسیم بندی به صورت زیر در می‌آید:

$$P_2 = \{\{A, E, F\}, \{\mathrm{B}\}, \{\mathrm{C}\}, \{D\}, \{\mathrm{G}\}\}$$

- و DFA با کمترین استیت به صورت زیر می‌باشد:



# ۷

- الف: نحوی(syntax)
- ب: معنایی(semantic)
- ج: نحوی(syntax)
- د: معنایی(semantic)
- ه: معنایی(semantic)

# ۸

- :cdccd
  - ✓ TOKEN_A :cd
  - ✓ TOKEN_B :ccd

TOKEN_ATOKEN_B

- :caccbd
  - ✓ TOKEN_A :cac
  - ✓ TOKEN_C :cb
  - ✓ d: هیچ matchingای رخ نمی‌دهد.

TOKEN_ATOKEN_Cd

- بنابراین برای این رشته با توجه به عبارات منظم و اولویت‌هایشان unmatching رخ خواهد داد.

# پاسخ سوالات عملی

# 1

```
import re

pattern_1 = '0*10*10*10*10*'
pattern_2 = '(1|0)*1(1|0){6,}|0*(111(1|0){3}|110(1|0){3}|101(10|01|11)(1|0))'
pattern_3 = '(0|1(10)*(0|11)(01*01|01*00(10)*(0|11))*1)*'
pattern_4 = '(0|00|10)*11(0|01)*(0)+(0|10)*11(01|00|0)*|(0|00|10)*111(01|00|0)*'

'''pattern_2: first_part:  (1|0)*1(1|0){6,} Includes all numbers greater than 64
              second_part: Includes all numbers between 41 to 63
   pattern_4: first_part: (0|00|10)*11(0|01)*(0)+(0|10)*11(01|00|0)*  Only two substrings of 11 are allowed
              second_part: (0|00|10)*111(01|00|0)*  If 111
'''
```

```
txt = r'00001001011100010100001'

r1 = re.compile(pattern_1)
x1 = r1.match(txt)
print("P1: Match!") if x1 and x1[0] == txt else print("P1: No Match!")

r2 = re.compile(pattern_2)
x2 = r2.match(txt)
print("P2: Match!") if x2 and x2[0] == txt else print("P2: No Match!")

r3 = re.compile(pattern_3)
x3 = r3.match(txt)
print("P3: Match!") if x3 and x3[0] == txt else print("P3: No Match!")

r4 = re.compile(pattern_4)
x4 = r4.match(txt)
print("P4: Match!") if x4 and x4[0] == txt else print("P4: No Match!")
```

```
%option noyywrap

%{
#include<stdio.h>
%}

%%
\/\/(.*) {};
\/\*(.*\n)*.*\*\/   {};
%%
```

```
int main()
{
   yyin=fopen("in.cpp","r");
   yyout=fopen("out.cpp","w");
   yylex();
   return 0;
}
```

```
%{
/* example illustrating the use of states in lex

   declare a state called INPUT using: %s INPUT

   enter a state using: BEGIN INPUT

   match a token only if in a certain state: <INPUT>\".*\"
*/
%}

%s INPUT
%s OUTPUT
```

```
%%

[ \t\n]+                ;
inputfile       BEGIN INPUT;
outputfile      BEGIN OUTPUT;
<INPUT>\".*\"   { BEGIN 0; ECHO; printf(" is the input file.\n"); }
<OUTPUT>\".*\"  { BEGIN 0; ECHO; printf(" is the output file.\n"); }

%%

int yywrap(){}

int main () {
  yylex();

}
```