Arizona State University
Computer Science and Engineering
CSE 340 - Fall 2018

Homework 2

Due Monday September 24 2018 by 11:59 PM

Please read the problems completely before asking questions.

1. **FIRST and FOLLOW** Calculate the FIRST and FOLLOW sets for the following grammar.

$$
\begin{array}{lll}
S & \rightarrow A\,B\,S & (1) \\
S & \rightarrow C\,S\,A & (2) \\
S & \rightarrow C\,B & (3) \\
A & \rightarrow A\,a & (4) \\
A & \rightarrow C\,S & (5) \\
C & \rightarrow c\,C & (6) \\
C & \rightarrow B & (7) \\
B & \rightarrow B\,b & (8) \\
B & \rightarrow \epsilon & (9)
\end{array}
$$

- For FIRST sets
  (a) Do an initialization pass by applying FIRST sets rules I and II.
  (b) Do successive passes, on the grammar rules in the order they are listed and apply to each grammar rule FIRST set rules III, then FIRST set rule IV then FIRST set rule V until there is no change in any of the FIRST sets.

  In your answer, you should show the FIRST sets after each pass. You do not have to use the notation I used in class to indicate the order in which the elements are added to the sets.
- You should calculate FOLLOW sets as follows:
  (a) Do an initialization pass by applying FOLLOW sets rule I
  (b) Do one pass on all grammar rules, in the order they are listed, and apply to each grammar rule FOLLOW set rules IV and V.
  (c) Do successive passes on all grammar rules in the order they are listed and apply to each grammar rule FOLLOW set rules II and III until there is no change in any of the FOLLOW sets.

In your answer, you should show the FOLLOW sets after each pass. You do not have to use the notation I used in class to indicate the order in which the elements are added to the sets.

Solution.

- FIRST sets

  (a) Do an initialization pass by applying FIRST sets rules I and II. We get:

  $$\begin{aligned}
  \text{FIRST}(\epsilon) &= \{\epsilon\}\\
  \text{FIRST}(a) &= \{a\}\\
  \text{FIRST}(b) &= \{b\}\\
  \text{FIRST}(c) &= \{c\}\\
  \text{FIRST}(S) &= \{\ \}\\
  \text{FIRST}(A) &= \{\ \}\\
  \text{FIRST}(B) &= \{\ \}\\
  \text{FIRST}(C) &= \{\ \}
  \end{aligned}$$

  (b) Do successive passes, on the grammar rules in the order they are listed and apply to each grammar rule FIRST set rules III, then FIRST set rule IV then FIRST set rule V until there is no change in any of the FIRST sets.

  i. At the end of the first pass we get the following:

  $$\begin{aligned}
  \text{FIRST}(\epsilon) &= \{\epsilon\}\\
  \text{FIRST}(a) &= \{a\}\\
  \text{FIRST}(b) &= \{b\}\\
  \text{FIRST}(c) &= \{c\}\\
  \text{FIRST}(S) &= \{\ \}\\
  \text{FIRST}(A) &= \{\ \}\\
  \text{FIRST}(B) &= \{\epsilon\}\\
  \text{FIRST}(C) &= \{c\}
  \end{aligned}$$

  ii. At the end of the second pass we get the following:

  $$\begin{aligned}
  \text{FIRST}(\epsilon) &= \{\epsilon\}\\
  \text{FIRST}(a) &= \{a\}\\
  \text{FIRST}(b) &= \{b\}\\
  \text{FIRST}(c) &= \{c\}\\
  \text{FIRST}(S) &= \{c\}\\
  \text{FIRST}(A) &= \{c\}\\
  \text{FIRST}(B) &= \{\epsilon, b\}\\
  \text{FIRST}(C) &= \{c, \epsilon\}
  \end{aligned}$$

  iii. At the end of the third pass we get the following:

  $$\begin{aligned}
  \text{FIRST}(\epsilon) &= \{\epsilon\}\\
  \text{FIRST}(a) &= \{a\}\\
  \text{FIRST}(b) &= \{b\}\\
  \text{FIRST}(c) &= \{c\}\\
  \text{FIRST}(S) &= \{c, b, \epsilon\}\\
  \text{FIRST}(A) &= \{c, b, \epsilon\}\\
  \text{FIRST}(B) &= \{\epsilon, b\}\\
  \text{FIRST}(C) &= \{c, \epsilon, b\}
  \end{aligned}$$

iv. At the end of the fourth pass we get the following:

$$\begin{aligned}
\text{FIRST}(\epsilon) &= \{\epsilon\} \\
\text{FIRST}(a) &= \{a\} \\
\text{FIRST}(b) &= \{b\} \\
\text{FIRST}(c) &= \{c\} \\
\text{FIRST}(S) &= \{c, \epsilon, b\} \\
\text{FIRST}(A) &= \{c, \epsilon, a, b\} \\
\text{FIRST}(B) &= \{\epsilon, b\} \\
\text{FIRST}(C) &= \{c, \epsilon, b\}
\end{aligned}$$

v. At the end of the fifth pass we get the following:

$$\begin{aligned}
\text{FIRST}(\epsilon) &= \{\epsilon\} \\
\text{FIRST}(a) &= \{a\} \\
\text{FIRST}(b) &= \{b\} \\
\text{FIRST}(c) &= \{c\} \\
\text{FIRST}(S) &= \{c, \epsilon, b, a\} \\
\text{FIRST}(A) &= \{c, \epsilon, a, b\} \\
\text{FIRST}(B) &= \{\epsilon, b\} \\
\text{FIRST}(C) &= \{c, \epsilon, b\}
\end{aligned}$$

vi. At the end of the sixth pass, there is no change and we stop.

- FOLLOW sets:

(a) Do an initialization pass by applying FOLLOW sets rule I

$$\begin{aligned}
\text{FOLLOW}(S) &= \{\$\} \\
\text{FOLLOW}(A) &= \{\ \} \\
\text{FOLLOW}(B) &= \{\ \} \\
\text{FOLLOW}(C) &= \{\ \}
\end{aligned}$$

(b) Do one pass on all grammar rules, in the order they are listed, and apply to each grammar rule FOLLOW set rules IV and V. At the end of this pass we get the following:

$$\begin{aligned}
\text{FOLLOW}(S) &= \{\$, c, a, b\} \\
\text{FOLLOW}(A) &= \{b, c, a\} \\
\text{FOLLOW}(B) &= \{c, b, a\} \\
\text{FOLLOW}(C) &= \{c, b, a\}
\end{aligned}$$

(c) Do successive passes on all grammar rules in the order they are listed and apply to each grammar rule FOLLOW set rules II and III until there is no change in any of the FOLLOW sets.

i. At the end of the first pass we get

$$\begin{aligned}
\text{FOLLOW}(S) &= \{\$, c, a, b\} \\
\text{FOLLOW}(A) &= \{b, c, a, \$\} \\
\text{FOLLOW}(B) &= \{c, b, a, \$\} \\
\text{FOLLOW}(C) &= \{c, b, a, \$\}
\end{aligned}$$

(d) At the end of the second pass there is no change and we stop

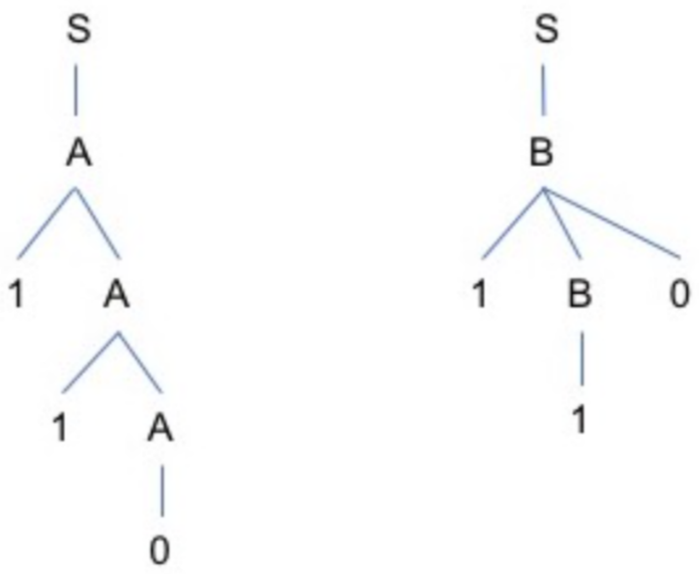
□

2. **Ambiguity**.

Show that the following grammar is ambiguous

$$\begin{array}{lll} S & \rightarrow A & (1) \\ S & \rightarrow B & (2) \\ A & \rightarrow 0\,A\,1 & (3) \\ A & \rightarrow 1\,A & (4) \\ A & \rightarrow 0 & (5) \\ B & \rightarrow 1\,B\,0 & (6) \\ B & \rightarrow 0\,B & (7) \\ B & \rightarrow 1 & (8) \end{array}$$

(a) Show that the following grammar is ambiguous by giving two parse trees for some input. The following are two different parse tress for the string 110



(b) Show that the following grammar is ambiguous by giving two leftmost derivations for some input (you can use the same input as in part 1). The following are two leftmost derivations for the input 110

i. $S \overset{(2)}{\Rightarrow} B \overset{(6)}{\Rightarrow} 1\,B\,0 \overset{(8)}{\Rightarrow} 1\,1\,B$

ii. $S \overset{(1)}{\Rightarrow} A \overset{(4)}{\Rightarrow} 1\,A \overset{(4)}{\Rightarrow} 1\,1\,A \overset{(5)}{\Rightarrow} 1\,1\,0$

3. **Parsing.** Consider the grammar

S → ABC

A → aABd | eBd

B → cBd | ε

C → eC | ε

(a) Show that the grammar has a predictive recursive descent parser

(b) Write the parser for the grammar. Your parser should detect syntax error as soon as possible by checking for FOLLOW set when reducing to ε

(c) Give a full execution trace for your parser from part (a) above on input e d c d.

FIRST(S) = {a, e}

FIRST(A) = {a, e}

FIRST(B) = {c, ε}

FIRST(C) = {e, ε}

FOLLOW(S) = {eof}

FOLLOW(A) = {c, d, e, eof}

FOLLOW(B) = {d, e, eof}

FOLLOW(C) = {eof}

(a) **Condition I of predictive parsing**

Rules for S: there is only one rule for S, so condition I is satisfied for S

Rules for A: A → a A B d and A → eBd

FIRST (a A B d) = {a}

FIRST(e B d) = {e}

FIRST (a A B d) ∩ FIRST(e B d) = ∅

So, condition I is satisfied for the rules of A.

Rules for B: B → c B d and B → ε

FIRST (c B d) = {c}

FIRST (ε) = {ε}

FIRST (c B d) ∩ FIRST (ε) = ∅

So, condition I is satisfied for the rules of B.

Rules for C: C → e C and C → ε

FIRST (e C) = {e}

FIRST (ε) = {ε}

FIRST (e C) ∩ FIRST (ε) = ∅

So, condition I is satisfied for the rules of C.

Condition II of predictive parsing

Nonterminal S: *ε* ∉ FIRST(S), so condition II holds to S

Nonterminal A: $\varepsilon \notin$ FIRST(A), so condition II holds to A

Nonterminal B: $\varepsilon \in$ FIRST(B), so we should have FIRST (B) ∩ FOLLOW (B) = ∅
FIRST(B) = {c, ε} FOLLOW(B) = {d, e, eof}
FIRST (B) ∩ FOLLOW (B) = ∅
So condition II holds for B

Nonterminal C: $\varepsilon \in$ FIRST(C), so we should have FIRST (C) ∩ FOLLOW (C) = ∅
FIRST(C) = {e, ε} FOLLOW(C) = {$}
FIRST (C) ∩ FOLLOW (C) = ∅
So condition II holds for C

This grammar satisfies both the conditions for predictive parsing and therefore it has a predictive parser.

(b) **Parser**

```
Parse_input() {
        Parse_S();
        tok = lexer.getToken();
        if(tok.tokenType == eof){
                print("valid match");
        } else {
                syntax_error();
        }
}
Parse_S(){
        tok = lexer.getToken();
        ttype = tok.token_type;
        if( (ttype == a-type) ||(ttype == e-type)) {
                lexer.ungetToken(tok)
                Parse_A();
                Parse_B();
                Parse_C();
                print("S -> A B C");
        } else {
                syntax_error();
        }
}
Parse_A(){
        tok = lexer.getToken();
        ttype = tok.token_type;
        if(ttype == a-type){
                Parse_A();
                Parse_B();
                tok = lexer.getToken();
                if(tok.token_type == d){
                        print("A -> a A B d");
                } else {
                        syntax_error();
                }
        } else if (ttype == e-type) {
                Parse_B();
                tok = lexer.getToken();
                if(tok.token_type == d-type){
                        print("A -> e B d");
                } else {
                        syntax_error();
                }
        } else {
                syntax_error();
        }
}

Parse_B(){
        tok = lexer.getToken();
        ttype = tok.token_type;
        if(ttype == c-type) {
                Parse_B();
                tok = lexer.getToken();
                if(tok.token_type == d-type){
                        print("B -> c B d");
                } else{
                        syntax_error();
                }
        } else if(ttype == d-type || ttype == e-type || ttype == eof){
                lexer.ungetToken(tok);
                print("B -> epsilon");
        } else {
                syntax_error();
        }
}

Parse_C(){
        tok = lexer.getToken();
        ttype = tok.token_type;
        if(ttype == e-type){
                Parse_C();
                print("C -> e C");
        } else if (ttype == eof) {
                lexer.ungetToken(tok);
                print("C -> epsilon");
        } else {
                syntax_error();
        }
}
```

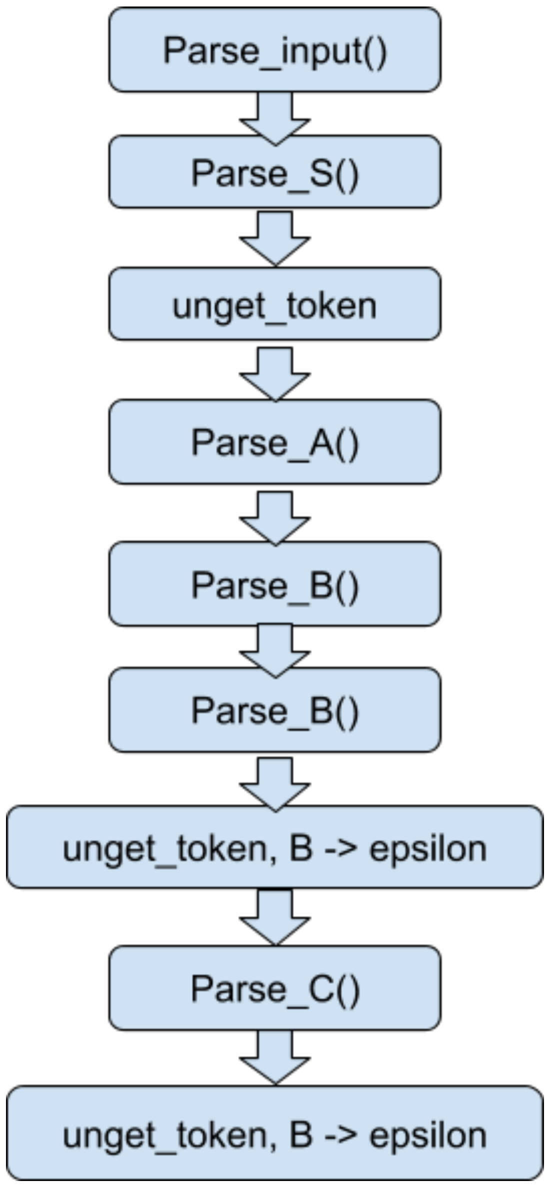
(c) **Execution trace on input e d c d:**

```
Parse_input() {
        Parse_S(){
                getToken()                                                   // e
                if( (ttype == a-type) ||(ttype == e-type)) {
                        ungetToken()                                         // e
                        Parse_A();
                                getToken()                                   // e
                                if(ttype == a-type)
                                else if (ttype == e-type) {
                                        Parse_B();
                                                getToken()                   // d
                                                if(ttype == c-type)
                                                else if(ttype ==d-type || ttype == e-type || ttype == eof){
                                                        ungetToken()         // d
                                                        print("B -> epsilon");
                                                }
                                        getToken()                           // d
                                        if(ttype == d-type){
                                                print("A -> e B d");
                                        }
                                }

                        Parse_B();
                                getToken()                                   // c
                                if(ttype == c-type){
                                        Parse_B();
                                                getToken()                   // d
                                                if(ttype == c-type)
                                                else if(ttype ==d-type || ttype == e-type || ttype == eof){
                                                        ungetToken()         // d
                                                        print("B -> epsilon");
```

```
                    }
                getToken()                                          // d
                if(ttype == d-type) {
                    print("B -> c B d");
                }
            }
        Parse_C();
            getToken()                                              // eof
            if(ttype == e-type)
            else if (ttype == eof) {
                ungetToken()                                        // eof
                print("C -> epsilon");
            }
        print("S -> A B C");
    }
}
getToken()                                                          // eof
if(ttype == eof) {
    print("valid match");
}
}
```

Call stack of functions:



4. **Useless Symbols.** Consider the following grammar G
S → AC | AE | AD
A → aA | B | a
B → bD
C → cB | A
D → dE | dB
D → aD | cB
(a) What are the generating symbols of this grammar?
(b) What are the reachable symbols of this grammar?

(c) What are the useless symbols of the grammar (note that this part has to be done by first eliminating non-generating symbols from G to obtain a new grammar G′ in which all symbols are generating, then eliminating non-reachable symbols from G′ to obtain the answer)?
Explain!

Answer:

a) **S, A and C as well as the terminal symbols** are generating. The following is the explanation

To identify the generating symbols, we follow the iterative algorithm discussed in class:
We create an array for all the symbols

0	1	2	3	4	5	6	7	8	9	10
“S”	“A”	“B”	“C”	“D”	“E”	“a”	“b”	“c”	“d”	“ε”

And a Boolean array that identifies which symbol indices in the above array are generating. Initially, all the values in the array are made False.

0	1	2	3	4	5	6	7	8	9	10
F	F	F	F	F	F	F	F	F	F	F

Next, we make all the indices True that correspond to terminal symbols and epsilon

0	1	2	3	4	5	6	7	8	9	10
F	F	F	F	F	F	T	T	T	T	T

Next, we start iterating over the rules
1) $S \rightarrow AC \mid AE \mid AD$
The entry corresponding to A has False value and the entry for S is not updated

2) $A \rightarrow aA \mid B \mid a$
A’s entry is updated to true because the all symbols in the right hand side of the $A \rightarrow a$ are generating

0	1	2	3	4	5	6	7	8	9	10
F	T	F	F	F	F	T	T	T	T	T

3) B → bD
The entry corresponding to D has False value and so the entry for B is not updated

4) C → cB | A
C’s entry is updated to true because the all symbols in the right hand side of the C → A are generating

0	1	2	3	4	5	6	7	8	9	10
F	T	F	T	F	F	T	T	T	T	T

5) D → dE | dB
The entries corresponding to E and B have False values and so the entry for D is not updated

6) D → aD | cB
The entries corresponding to D and B have False values and so the entry for D is not updated

Since, the table changed, we iterate again over all the rules

1) S → AC | AE | AD
S’s entry is updated to true because the all symbols in the right hand side of the S → AC are generating

0	1	2	3	4	5	6	7	8	9	10
T	T	F	T	F	F	T	T	T	T	T

2) A → aA | B | a
entry for A is already true, so it will not change

3) B → bD
The entry corresponding to D has False value and so the entry for B is not updated

4) C → cB | A
entry for C is already true, so it will not change

5) D → dE | dB
The entries corresponding to E and B have False values and so the entry for D is not updated

6) D → aD | cB
The entries corresponding to D and B have False values and so the entry for D is not updated

Since, the table changed, we iterate again over all the rules

1) S → AC | AE | AD
entry for S is already true, so it will not change

2) A → aA | B | a
entry for A is already true, so it will not change

3) B → bD
The entry corresponding to D has False value and so the entry for B is not updated

4) C → cB | A
entry for C is already true, so it will not change

5) D → dE | dB
The entries corresponding to E and B have False values and so the entry for D is not updated

6) D → aD | cB
The entries corresponding to D and B have False values and so the entry for D is not updated

Since no change was made to the Boolean array, the algorithm stops.

The indices, which have value True, represent the corresponding symbols that are generating.
Hence, the generating symbols are S, A and C in addition to the terminals

b) All symbols are reachable

To identify the reachable symbols, we follow the iterative algorithm discussed in class:
We create an array for all the symbols

0	1	2	3	4	5	6	7	8	9
“S”	“A”	“B”	“C”	“D”	“E”	“a”	“b”	“c”	“d”

And a Boolean array that identifies which symbol indices in the above array are reachable. Initially, all the values in the array are made False.

0	1	2	3	4	5	6	7	8	9
F	F	F	F	F	F	F	F	F	F

1. S is reachable by definition

0	1	2	3	4	5	6	7	8	9
T	F	F	F	F	F	F	F	F	F

2. S → A C implies that A and C are reachable because S is reachable

0	1	2	3	4	5	6	7	8	9
T	T	F	T	F	F	F	F	F	F

3. S → A E implies that A and E are reachable because S is reachable

0	1	2	3	4	5	6	7	8	9
T	T	F	T	F	T	F	F	F	F

4. S → A D implies that A and D are reachable because S is reachable

0	1	2	3	4	5	6	7	8	9
T	T	F	T	T	T	F	F	F	F

5. A → a A implies that a is reachable because A is reachable

0	1	2	3	4	5	6	7	8	9
T	T	F	T	T	T	T	F	F	F

6. A → B implies that B is reachable because A is reachable

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	F	F	F

7. A → a implies that a is reachable because A is reachable. There is no change to the boolean array.
8. B → bD implies that b and D are reachable because B is reachable

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	T	F	F

9. C → c B implies that c and B are reachable because C is reachable

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	T	T	F

10. C → A implies that A is reachable because C is reachable. There is no change to the boolean array.
11. D → d E implies that d and E are reachable because D is reachable

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	T	T	T

12. D → d B implies that d and B are reachable because D is reachable. There is no change to the boolean array.
13. D → a D implies that a and D are reachable because D is reachable. There is no change to the boolean array.
14. D → c B implies that c and B are reachable because D is reachable. There is no change to the boolean array.

The boolean array after first pass :

0	1	2	3	4	5	6	7	8	9
T	T	T	T	T	T	T	T	T	T

After the second pass, no changes will be made to the array, so we terminate after second pass. All the symbols are reachable.

c) To determine useless symbols, we start by eliminating rules that contain non-generating symbols. We know that B, D and E are non-generating. This means that we should eliminate the rules

S → AE | AD
A → B
B → bD

$C \rightarrow cB$
$D \rightarrow dE \mid dB$
$D \rightarrow aD \mid cB$

This only leaves the rules

$S \rightarrow AC$
$A \rightarrow aA \mid a$
$C \rightarrow A$

Now, we determine all reachable symbols. S is reachable by definition.
$S \rightarrow A\ C$ implies that A and C are reachable because S is reachable.
$A \rightarrow aA$ implies that a and A are reachable because A is reachable.
$A \rightarrow a$ implies that a is reachable because A is reachable.
$C \rightarrow A$ implies that A is reachable because C is reachable.

So the reachable symbols are S, A and C.

Thus the symbols B, D and E are useless symbols.