CSE340 FALL 2021 - HOMEWORK 2
Solutions

Due: Friday October 1, 2021 by 11:59 PM on Canvas

All submissions should be typed, no exceptions.

1. Consider the grammar.

S → A B C D	(1)
A → A B &	(2)
A → C D	(3)
C → A c	(4)
C → ε	(5)
B → b B !	(6)
B → C	(7)
D → d D	(8)
D → ε	(9)

where S, A, B, C and D are the non-terminals, S is the start symbol, and b, c , d '&' and '!' are the terminals.

1.1. **FIRST sets**. Do the following

1.1.1. Do an initialization pass by applying FIRST sets rules I and II. Show the resulting FIRST sets

FIRST(ε) = {ε}

FIRST(b) = {b}

FIRST(c) = {c}

FIRST(d) = {d}

FIRST(&) = {&}

FIRST(!) = {!}

FIRST(S) = {}

FIRST(A) = {}

FIRST(B) = {}

FIRST(C) = {}

FIRST(D) = {}

Explanation: rules I and II apply only to epsilon and terminals, so this should be self-explanatory

1.1.2. Do one pass on the grammar rules in the order in which they are listed as follows. For each grammar rules, apply FIRST set rule III, then apply FIRST set rule IV then apply FIRST set rule V. Show the resulting FIRST sets after this one pass.

FIRST(ε) = {ε}

FIRST(b) = {b}

FIRST(c) = {c}

FIRST(d) = {d}

FIRST(&) = {&}

FIRST(!) = {!}

FIRST(S) = {}

FIRST(A) = {}

FIRST(B) = {b[2], ε[3]}

FIRST(C) = {ε[1]}

FIRST(D) = {d[4], ε[5]}

Explanation: We go over each grammar rule in order from (1) to (9). When we consider a rule, we apply rules II, IV and V in that order and we consult the FIRST sets as they evolve in applying the rules. The numbers above indicate the order in which elements are added

1.1.3. Do a second pass on the grammar rules in the order in which they are listed as follows. For each grammar rules, apply FIRST set rule III, then apply FIRST set rule IV then apply FIRST set rule V. Show the resulting FIRST sets after this one pass.

FIRST(ε) = {ε}

FIRST(b) = {b}

FIRST(c) = {c}

FIRST(d) = {d}

FIRST(&) = {&}

FIRST(!) = {!}

FIRST(S) = {}

FIRST(A) = {d[6], ε[7]}

FIRST(B) = {b[2], ε[3], d[10], c[10]}

FIRST(C) = {ε[1], d[8], c[9]}

FIRST(D) = {d[4], ε[5]}

Explanation: We go over each grammar rule in order from (1) to (9). When we consider a rule, we apply rules II, IV and V in that order and we consult the FIRST sets as they evolve in applying the rules. The numbers above indicate the order in which elements are added

1.1.4. Show the final result of the calculation of the FIRST sets

FIRST(ε) = {ε}

FIRST(b) = {b}

FIRST(c) = {c}

FIRST(d) = {d}

FIRST(&) = {&}

FIRST(!) = {!}

FIRST(S) = {d[11], b[12], c[12], ε[13],&[18]}

FIRST(A) = { d[6], ε[7], b[14], c[14], &[15]}

FIRST(B) = {b[2], ε[3], d[10], c[10], &[17]}

FIRST(C) = { ε[1], d[8], c[9], b[16], &[16]}

FIRST(D) = { d[4], ε[5]}

Explanation: We go over each grammar rule in order from (1) to (9). When we consider a rule, we apply rules II, IV and V in that order and we consult the FIRST sets as they evolve in applying the rules. The numbers above indicate the order in which elements are added. Multiple passes were needed to get the result.

1.2. **FOLLOW sets**. Do the following

1.2.1. Do an initialization pass by applying FOLLOW set rules I. Then do one pass by applying FOLLOW sets rules IV, and V (adding FIRST to FOLLOW). Show the resulting FOLLOW sets

FOLLOW(S) = {$[1]}

FOLLOW(A) = {b[2], d[2], c[2], &[2]}

2: obtained from grammar rule (1) by adding FIRST(B) – {Ɛ} to FOLLOW(A)

FOLLOW(B) = {d[3], c[3], b[3], &[3], ![4]}

3: obtained from grammar rule (1) by adding FIRST(C) – {Ɛ} to FOLLOW(B)

4: from grammar rule (6) by adding FIRST(!) - {Ɛ} to FOLLOW(B)

FOLLOW(C) = {d[5]}

5: obtained from grammar rule (1) by adding FIRST(D) – {Ɛ} to FOLLOW(C)

FOLLOW(D) = {}

1.2.2. Do one pass on the grammar rules in the order in which they are listed as follows. For each grammar rules, apply FOLLOW set rule II, then apply FOLLOW set rule III. Show the resulting FOLLOW sets after this one pass.

FOLLOW(S) = {$[1]}

FOLLOW(A) = {b[2], d[2], c[2], &[2], $[9]}

9: obtained from grammar rule (1) by adding FOLLOW(S) to FOLLOW(A) because Ɛ is in FIRST(D) and Ɛ is in FIRST(C) and Ɛ is in FIRST(B):

S → A B C D

FOLLOW(B) = {d[3], c[3], b[3], &[3], ![4], $[8]}

8: obtained from grammar rule (1) by adding FOLLOW(S) to FOLLOW(B) because Ɛ is in FIRST(D) and Ɛ is in FIRST(C):

S → A B C D

FOLLOW(C) = {d[5], $[7], b[11], d[11], c[11], &[11], ![12]}

7: obtained from grammar rule (1) by adding FOLLOW(S) to FOLLOW(C) because Ɛ is in FIRST(D):

S → A B C D

11: obtained from grammar rule (3) by adding FOLLOW(A) to FOLLOW(C) which we can do because Ɛ is in FIRST(D):

A → C D

12: obtained from grammar rule (7) by adding FOLLOW(B) to FOLLOW(C):

B → C

FOLLOW(D) = {$[6], b[10], d[10], c[10], &[10]}

6: obtained from grammar rule (1) by adding FOLLOW(S) to FOLLOW(D)

S → A B C D

10: from grammar rule (3) by adding FOLLOW(A) to FOLLOW(D)

A → C D

1.2.3. Do a second pass on the grammar rules in the order in which they are listed as follows. For each grammar rules, apply FOLLOW set rule II, then apply FOLLOW set rule III. Show the resulting FOLLOW sets after this one pass.

There is no change in this second pass

FOLLOW(S) = {$[1]}

FOLLOW(A) = {b[2], d[2], c[2], &[2], $[9]}

FOLLOW(B) = {d[3], c[3], b[3], &[3], ![4], $[8]}

FOLLOW(C) = {d[5], $[7], b[11], d[11], c[11], &[11], ![12]}

FOLLOW(D) = {$[6], b[10], d[10], c[10], &[10]}

1.2.4. Show the final result of the calculation of the FOLLOW sets

FOLLOW(S) = {$[1]}

FOLLOW(A) = {b[2], d[2], c[2], &[2], $[9]}

FOLLOW(B) = {d[3], c[3], b[3], &[3], ![4], $[8]}

FOLLOW(C) = {d[5], $[7], b[11], d[11], c[11], &[11], ![12]}

FOLLOW(D) = {$[6], b[10], d[10], c[10], &[10]}

See homework 2 from last semester for a similar problem

2. Consider the grammar

S → a S b | c

A → C S | C S E

C → D B | B

B → a C a | ε

D → d | ε

Show that this grammar does not have a recursive descent predictive parser. To get full credit you should show, **for each non-terminal**, all the conditions of predictive parsing that are not satisfied by the rules of the non-terminal. It is not enough to show that for some non-terminal the rules are not satisfied.

You do not need to show the conditions of predictive parsing that are satisfied.

You will need to calculate FIRST and FOLLOW sets but you do not need to show how you calculated them. You will need to explicitly show which conditions of predictive parsing are not satisfied.

Note. As Emilia pointed out, if S is the start symbol, which it is, that would make A and all other symbols useless because there is no derivation from S in which they appear, so A is the more natural start symbol. In fact, if we are only interested in parsing S, the grammar has a predictive parser! Also, the typed notes state that the condition for the existence or non-existence of a predictive parser assume that there are no useless symbols, which is really not the case here. This was not meant as a trick question and I did not notice the situation when I asked the question. I was changing the rules for S continuously as I was writing it and forgot to include A in one of the rules for S ☹. So, the answer given ignores the fact that there are useless symbols, but in grading the homework if the issue is brought up in the answer, that would be considered.

The FIRST sets are as follows:

FIRST(ε) = {ε}

FIRST(a) = {a}

FIRST(b) = {b}

FIRST(c) = {c}

FIRST(d) = {d}

FIRST(E) = {E}

The FOLLOW sets are as follows:

FOLLOW(S) = {$, b, E}

FOLLOW(A) = {}

FOLLOW(B) = {a, c}

FOLLOW(C) = {a, c}

FOLLOW(D) = {a, c}

FIRST(S) = {a, c}

FIRST(A) = {d, a, c}

FIRST(B) = {a, ε}

FIRST(C) = {d, a, ε}

FIRST(D) = {d, ε}

A grammar has a predictive parser if and only if the following two conditions hold

1. For every non-terminal A and any two rules, A -> $\boldsymbol{\alpha}$ or A -> $\boldsymbol{\beta}$, FIRST ($\boldsymbol{\alpha}$) ∩ FIRST($\boldsymbol{\beta}$) = ∅
2. If a non-terminal A derives ε (A ⇒* ε), then FIRST (A) ∩ FOLLOW(A) = ∅

Check Condition 1:

Non-terminal S:

S → a S b

S → c

FIRST(a S b) = {a}

FIRST(c) = {c}

FIRST(a S b) ∩ FIRST(c) = ∅

So, condition 1 is satisfied for S.

Non-terminal A:

A → C S

A → C S E

FIRST(C S) = {d, a, c}

FIRST(C S E) = {d, a, c}

FIRST(C S) ∩ FIRST(C S E) ≠ ∅

So, condition 1 is not satisfied for A.

Non-terminal B:

B → a C a

B → ε

FIRST(a C a) = {a}

FIRST(ε) = {ε}

FIRST(aCa) ∩ FIRST(ε) = ∅

So, condition 1 is satisfied for B.

Non-terminal C:

C → D B

C → B

FIRST(D B) = {d, b ,ε}

FIRST(B) = {a, ε}

FIRST(D B) ∩ FIRST(B) ≠ ∅

So, condition 1 is not satisfied for C.

Non-terminal D:

D → d

$D \rightarrow \varepsilon$
$FIRST(d) = \{d\}$
$FIRST(\varepsilon) = \{\varepsilon\}$
$FIRST(d) \cap FIRST(\varepsilon) = \emptyset$
So, condition 1 is satisfied for D.

Condition 1 is not satisfied for non-terminals A and C.

Check Condition 2: We only need to check condition 2 for the non-terminals that generate epsilon. They are B, C and D

$FIRST(B) \cap FOLLOW(B) = \{a\} \neq \emptyset$
$FIRST(C) \cap FOLLOW(C) = \{a\} \neq \emptyset$
$FIRST(D) \cap FOLLOW(D) = \emptyset$

Condition 2 is not satisfied by non-terminals B and C which generate epsilon.

Since this grammar does not satisfy both conditions 1 and 2, this grammar does not have a predictive recursive descent parser.

3. Consider the grammar

$S \rightarrow s\ S\ t \mid A\ k\ B\ C$

$A \rightarrow f\ A\ g \mid E\ D$

$B \rightarrow b\ B \mid g$

$C \rightarrow c \mid E$

$D \rightarrow d\ D \mid \varepsilon$

$E \rightarrow e\ E \mid \varepsilon$

3.1. Show that the grammar has a predictive recursive descent parser. You should show that the conditions of predictive parsing apply for every non-terminal.

The FIRST sets are as follows:

$FIRST(\varepsilon) = \{\varepsilon\}$
$FIRST(b) = \{b\}$
$FIRST(c) = \{c\}$
$FIRST(d) = \{c\}$
$FIRST(e) = \{e\}$
$FIRST(f) = \{f\}$
$FIRST(g) = \{g\}$
$FIRST(k) = \{k\}$
$FIRST(s) = \{s\}$
$FIRST(t) = \{t\}$
$FIRST(S) = \{s, f, e, k, d\}$
$FIRST(A) = \{f, e, d, \varepsilon\}$

The FOLLOW sets are as follows:

$FOLLOW(S) = \{\$, t\}$
$FOLLOW(A) = \{k, g\}$
$FOLLOW(B) = \{c, e, \$, t\}$
$FOLLOW(C) = \{\$, t\}$
$FOLLOW(D) = \{k, g\}$
$FOLLOW(E) = \{\$, d, k, g, t\}$

FIRST(B) = {b, g}

FIRST(C) = {c, e, ε}

FIRST(D) = {d, ε}

FIRST(E) = {e, ε}

A grammar has a predictive parser if and only if the following two conditions hold

1. For every non-terminal A and any two rules, A -> $\boldsymbol{\alpha}$ or A -> $\boldsymbol{\beta}$, FIRST ($\boldsymbol{\alpha}$) ∩ FIRST($\boldsymbol{\beta}$) = ∅
2. If a non-terminal A derives ε (A ⇒* ε), then FIRST (A) ∩ FOLLOW(A) = ∅

Check Condition 1:

Non-terminal S:

S → sSt

S → AkBC

FIRST(sSt) = {s}

FIRST(AkBC) = {f, e, d, k}

FIRST (sSt) ∩ FIRST(AkBC) = ∅

So, condition 1 is satisfied for S.

Non-terminal A:

A → fAg

A → ED

FIRST(fAg) = {f}

FIRST(ED) = {e, d, ε}

FIRST(fAg) ∩ FIRST(ED) = ∅

So, condition 1 is satisfied for A.

Non-terminal B:

B → bB

B → g

FIRST(bB) = {b}

FIRST(g) = {g}

FIRST(bB) ∩ FIRST(g) = ∅

So, condition 1 is satisfied for B.

Non-terminal C:

C → c

C → E

FIRST(c) = {c}

FIRST(E) = {e, ε}

FIRST(c) ∩ FIRST(E) = ∅

So, condition 1 is satisfied for C.

Non-terminal D:

D → dD

D → ε

FIRST(dD) = {d}

FIRST(ε) = {ε}
FIRST(dD) ∩ FIRST(ε) = ∅
So, condition 1 is satisfied for D.

Non-terminal E:

E → eE
E → ε
FIRST(eE) = {e}
FIRST(ε) = {ε}
FIRST(eE) ∩ FIRST(ε) = ∅
So, condition 1 is satisfied for E

Condition 1 is satisfied for all non-terminals.

Check Condition 2: We only need to check condition 2 for the non-terminals that generate epsilon. They are A, C, D and E

FIRST(A) ∩ FOLLOW(A) = ∅
FIRST(C) ∩ FOLLOW(C) = ∅
FIRST(D) ∩ FOLLOW(D) = ∅
FIRST(E) ∩ FOLLOW(E) = ∅

Condition 2 is satisfied for all non-terminals that generate epsilon.

Since conditions 1 and 2 are both, this grammar has a predictive recursive descent parser.

3.2. Write parse_input()

```
void parse_input()
{
        parse_S();
        lexer.expect(EOF);
}
```

3.3. Write parse_S()

```
void parse_S()
{       // S -> sSt      FIRST(sSt) = {s}
        // S -> AkBC  FIRST(AkBC) ={f, e, d, k}

        Token t = lexer.peek(1);
        if (t.token_type == s_type)
        {
                lexer.expect(s_type);
                parse_S();
                lexer.expect(t_type);
        }
        else if ((t.token_type == f_type) || (t.token_type == e_type)||
                (t.token_type == d_type) || (t.token_type == k_type))
        {
                parse_A();
                lexer.expect(k_type);
                parse_B();
```

```
                parse_C();
        }
        else
                syntax_error();
}
```

3.4. Write parse_A().

```
void parse_A()
{       // A -> fAg     FIRST(fAg) = {f}
        // A -> ED      FIRST(ED) ={e, d, ε}
        // FOLLOW(A) = {k, g}

        Token t = lexer.peek(1);
        if (t.token_type == f_type)
        {
                lexer.expect(f_type);
                parse_A();
                lexer.expect(g_type);
        }
        else if ( (t.token_type == e_type) || (t.token_type == d_type) )  // FIRST(ED)
        {
                parse_E();
                parse_D();
        }
        else if ((t.token_type == k_type) || (t.token_type == g_type)) // FOLLOW(A)
        {
                parse_E();
                parse_D();
        }
        else
                syntax_error();
}
```

3.5. Give a **full execution** trace for your parser from part 3.2 above on input s s f g k g t t

trace should show the correct sequence of calls including calls to functions that you were not asked to write in parts 3.2, 3.3 and 3.4 above. The trace should show the sequence of function calls like parse_X() where X is a non-terminal and expect(ttype), where ttype is a token type.

Your parse functions should follow the general model of predictive parser that we saw in class. In particular, for non-terminals that can generate ε, the parser should check the FOLLOW set before choosing to parse the righthand side that generates ε.

Answer is on the next page

```
parse_input()
        parse_S()
                lexer.peek(1)                                   // next token is s
                lexer.expect(s_type)                            //s is consumed
                parse_S()
                        lexer.peek(1)                           // next token is s
                        lexer.expect(s_type)                    //s is consumed
                        parse_S()
                                lexer.peek(1)                   // next token is f
                                parse_A();
                                        lexer.peek(1)           //next token is f
                                        lexer.expect(f_type);   //f is consumed
                                        parse_A();
                                                lexer.peek(1)   //next token is g
                                                                // A ⇒ E D ⇒* epsilon
                                                parse_E()
                                                        lexer.peek(1);  // next token is g
                                                        return;         // E is epsilon
                                                parse_D()
                                                        lexer.peek(1);  // next token is g
                                                        return;         // D is epsilon
                                        lexer.expect(g_type);   //g is consumed
                                lexer.expect(k_type);           // k is consumed
                                parse_B();
                                        lexer.peek(1)           // next token is g
                                        lexer.expect(g_type)    //g is consumed
                                parse_C();
                                        lexer.peek(1)           //next token is t
                                        parse_E();              // C ⇒ E ⇒ epsilon
                                                lexer.peek(1);
                                                return;
                        lexer.expect(t_type)                    //t is consumed
                lexer.expect(t_type)
        lexer.expect(EOF);                                      //t is consumed
```

4. We say that a symbol of a grammar is useless if it does not appear in the derivation of a string (possibly empty) of terminals. Formally, *A is useful* if there exists a derivation S ⇒* x A y ⇒* w where w is a string of terminals. *A is useless* if it is not useful. Note that a useful symbol can be either a terminal or a non-terminal. Also, a useless symbol can be a terminal or a non-terminal.

Example.

S → S A B | a B C

A → A S | B E

B → a B | A | b

C → c | E

F → c | E

E → e E

1. F is useless because there is no derivation starting with S in which F appears.
2. E is useless because it cannot derive a string of terminals, so there can be no derivation of a string of terminals starting from S in which E appears.
3. e is useless because there is no derivation of a string of terminals starting from S in which e appears
4. A is useless because it cannot derive a string of terminals. One rule is recursive (A → A S), so we cannot hope of getting a string of terminals from A by only using the rule A → A S. The other rule for A also cannot derive a string of terminals because it contains E.
5. S is not useless because it appears in the derivation S ⇒ a B C ⇒ a b C ⇒ a b c
6. B is not useless because it appears in the derivation S ⇒ a B C ⇒ a b C ⇒ a b c
7. C is not useless because it appears in the derivation S ⇒ a B C ⇒ a b C ⇒ a b c

Consider the grammar

S → A B G | D E F

A → a B | B D

B → b D | A E

E → e E | C E

C → c A | E | c

D → d | F g

F → f F | G f

G → g G | F g

Which are the useful symbols of this grammar? For every useful symbol, give a derivation that shows that the symbol is useful. If a derivation shows multiple symbols to be useful, you can give the derivation and state which symbols are useful according to the derivation.

This grammar has no useful symbols. Useful symbols are symbols that appear in derivations of strings of terminals (including the empty string). In particular, since they appear in a derivation of a string of terminals, useful symbols must themselves be able to generate strings of terminals (being able gto generate a string of terminals is necessary but not sufficient for a symbol to be useful).

In this grammar all non-terminals cannot generate strings of terminals and it follows that there are no useful symbols because S cannot generate a string of terminals. To see why this is the case, we consider all non-terminals.

For example, see what happens if we try to start at G and derive a string of terminals.

G ⇒ g G ⇒ g F g ⇒ g f F g ⇒ g f G f g ⇒ g f g G f g ⇒ g f g F g f g ⇒ g f g G f g f g ⇒ …...

No matter what we try, at each step we will have either F or G in the string and we cannot get rid of them. Similarly if we start with F.

Since F and G cannot derive strings of terminals, it follows that S itself cannot derive a string of terminals because one of the rules for S has G on the RHS and the other rule has F on the RHS, so starting from S we will not be able generate a string of terminals because F and G cannot generate strings of terminals.

All other symbols are useless because S is useless!