

CSE340 SPRING 2020 - HOMEWORK 2 SOLUTION
Due : Wednesday February 2 2020 by 11:59 PM on Canvas

All submissions should be **typed**, no exceptions.

Note. This solution can add explanations for some problems even if the problem statement does not ask for the explanation.

1. Consider the grammar

$$S \rightarrow B A g \quad (1)$$

$$A \rightarrow f A b \quad (2)$$

$$A \rightarrow (C A B) \quad (3)$$

$$A \rightarrow C \quad (4)$$

$$B \rightarrow a B c \quad (5)$$

$$B \rightarrow \epsilon \quad (6)$$

$$C \rightarrow c C \quad (7)$$

$$C \rightarrow \epsilon \quad (8)$$

where S, A, B, C and D are the non-terminals, S is the start symbol, and a, b, c, f, g, '(' and ')' are the terminals.

Solution:

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.

$$\text{FIRST}(a) = \{a\}$$

$$\text{FIRST}(b) = \{b\}$$

$$\text{FIRST}(c) = \{c\}$$

$$\text{FIRST}(f) = \{f\}$$

$$\text{FIRST}(g) = \{g\}$$

$$\text{FIRST}('(') = \{ '(' \}$$

$$\text{FIRST}(')') = \{ ')' \}$$

$$\text{FIRST}(\epsilon) = \{\epsilon\}$$

$$\text{FIRST}(S) = \{\}$$

$$\text{FIRST}(A) = \{\}$$

$$\text{FIRST}(B) = \{\}$$

$$\text{FIRST}(C) = \{\}$$

1.1.2 Do one pass on the grammar rules in the order in which they are listed as follows. For each grammar rule, 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.

$\text{FIRST}(a) = \{ a \}$
 $\text{FIRST}(b) = \{ b \}$
 $\text{FIRST}(c) = \{ c \}$
 $\text{FIRST}(f) = \{ f \}$
 $\text{FIRST}(g) = \{ g \}$
 $\text{FIRST}('(') = \{ '(' \}$
 $\text{FIRST}(')') = \{ ')' \}$
 $\text{FIRST}(\epsilon) = \{ \epsilon \}$
 $\text{FIRST}(S) = \{ \}$
 $\text{FIRST}(A) = \{ f, '(' \}$
 $\text{FIRST}(B) = \{ a, \epsilon \}$
 $\text{FIRST}(C) = \{ c, \epsilon \}$

1.1.3 Show the final result of the calculation of the FIRST sets

$\text{FIRST}(a) = \{ a \}$
 $\text{FIRST}(b) = \{ b \}$
 $\text{FIRST}(c) = \{ c \}$
 $\text{FIRST}(f) = \{ f \}$
 $\text{FIRST}(g) = \{ g \}$
 $\text{FIRST}('(') = \{ '(' \}$
 $\text{FIRST}(')') = \{ ')' \}$
 $\text{FIRST}(\epsilon) = \{ \epsilon \}$
 $\text{FIRST}(S) = \{ a, f, '(', c, g \}$
 $\text{FIRST}(A) = \{ f, '(', c, \epsilon \}$
 $\text{FIRST}(B) = \{ a, \epsilon \}$
 $\text{FIRST}(C) = \{ c, \epsilon \}$

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. Show the resulting FOLLOW sets.

$\text{FOLLOW}(S) = \{ \$ \}$
 $\text{FOLLOW}(A) = \{ g, b, a, ')' \}$
 $\text{FOLLOW}(B) = \{ f, '(', c, g, ')' \}$
 $\text{FOLLOW}(C) = \{ f, '(', c, a, ')' \}$

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

$\text{FOLLOW}(S) = \{ \$ \}$
 $\text{FOLLOW}(A) = \{ g, b, a, ')' \}$

$\text{FOLLOW}(B) = \{ f, '(', c, g, ')' \}$
 $\text{FOLLOW}(C) = \{ f, '(', c, a, ')', g, b \}$

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

$\text{FOLLOW}(S) = \{ \$ \}$
 $\text{FOLLOW}(A) = \{ g, b, a, ')' \}$
 $\text{FOLLOW}(B) = \{ f, '(', c, g, ')' \}$
 $\text{FOLLOW}(C) = \{ f, '(', c, a, ')', g, b \}$

2. Consider the grammar

$S \rightarrow A B C$
 $A \rightarrow a A b \mid c \mid B$
 $B \rightarrow b B c \mid D$
 $C \rightarrow b B c \mid \epsilon$
 $D \rightarrow d \mid \epsilon$

Show that this grammar does not have a recursive descent predictive parser. To get full credit you should show all the conditions of predictive parsing that are not satisfied by this grammar. 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.

Solution 2:

The first and follow sets are as follows:

FIRST SETS:

FIRST(a)	{ a }
FIRST(b)	{ b }
FIRST(c)	{ c }
FIRST(ϵ)	{ ϵ }
FIRST(d)	{ d }
FIRST(A)	{ a, c, b, d, ϵ }
FIRST(B)	{ b, d, ϵ }
FIRST(C)	{ b, ϵ }

FIRST(D)	{ d, ϵ }
FIRST(S)	{ a, c, b, d, ϵ }

FOLLOW SETS:

FOLLOW(S)	{ EOF }
FOLLOW(A)	{ b, d, EOF }
FOLLOW(B)	{ c, b, d, EOF }
FOLLOW(C)	{ EOF }
FOLLOW(D)	{ c, b, d, EOF }

To prove that the given grammar has a predictive recursive descent parser, it should satisfy the following conditions

1. If $A \rightarrow \alpha$ and $A \rightarrow \beta$ are two grammar rules, then $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
2. If $\epsilon \in FIRST(A)$ then $FIRST(A) \cap FOLLOW(A) = \emptyset$

Condition 1:

If a non-terminal symbol appears on the left-hand side of only one rule, condition 1 is immediately satisfied. We consider non-terminals that have more than one rule. These non-terminals are A, B, C, D.

a) Non-terminal **A**

$A \rightarrow aAb$

$A \rightarrow c$

$A \rightarrow B$

$FIRST(aAb) = \{a\}$

$FIRST(c) = \{c\}$

$FIRST(B) = \{b, d, \epsilon\}$

$FIRST(aAb) \cap FIRST(c) = \emptyset$

$FIRST(aAb) \cap FIRST(B) = \emptyset$

$FIRST(B) \cap FIRST(c) = \emptyset$

Condition 1 is satisfied for A.

b) Non-terminal **B**

$B \rightarrow bBc$

$B \rightarrow D$

$FIRST(bBc) = \{b\}$

$FIRST(D) = \{d, \epsilon\}$

$FIRST(bBc) \cap FIRST(D) = \emptyset$

Condition 1 is satisfied for B

c) Non-terminal **C**

$C \rightarrow bBc$

$C \rightarrow \epsilon$

$FIRST(bBc) = \{b\}$

$FIRST(\epsilon) = \{\epsilon\}$

$FIRST(bBc) \cap FIRST(\epsilon) = \emptyset$

Condition 1 is satisfied for C

d) Non-terminal **D**

$D \rightarrow d$

$D \rightarrow \epsilon$

$FIRST(d) = \{d\}$

$FIRST(\epsilon) = \{\epsilon\}$

$FIRST(d) \cap FIRST(\epsilon) = \emptyset$

Condition 1 is satisfied for D

We have shown that the grammar satisfies the first condition for predictive parsing. Now, we consider the second condition.

Condition 2: If $\epsilon \in FIRST(A)$ then $FIRST(A) \cap FOLLOW(A) = \emptyset$

For every non-terminal A such that $\epsilon \in FIRST(A)$, we need to show that $FIRST(A) \cap FOLLOW(A) = \emptyset$. The non-terminals that have ϵ in their FIRST sets are S, A, B, C, and D.

$FIRST(S) \cap FOLLOW(S) = \{a, c, b, d, \epsilon\} \cap \{EOF\} = \emptyset$

$FIRST(A) \cap FOLLOW(A) = \{a, c, b, d, \epsilon\} \cap \{b, d, EOF\} = \{b, d\}$ which is not empty

$FIRST(B) \cap FOLLOW(B) = \{b, d, \epsilon\} \cap \{c, b, d, EOF\} = \{b, d\}$ which is not empty

$$\text{FIRST}(C) \cap \text{FOLLOW}(C) = \{b, \epsilon\} \cap \{\text{EOF}\} = \emptyset$$

$$\text{FIRST}(D) \cap \text{FOLLOW}(D) = \{d, \epsilon\} \cap \{c, b, d, \text{EOF}\} = \{d\} \text{ which is not empty}$$

As highlighted above Condition 2 is not satisfied by Non-terminals A,B and D. Therefore, this grammar does not satisfy condition 2. Given that the grammar does not satisfy both conditions for the existence of a predictive recursive descent parser, the grammar does not have a predictive recursive descent parser.

3. Consider the grammar

$$S \rightarrow A b C \mid D c A$$

$$A \rightarrow a A d \mid E \mid f$$

$$E \rightarrow e \mid \epsilon$$

$$C \rightarrow c \mid \epsilon$$

$$D \rightarrow d D \mid \epsilon$$

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.

Solution 3.1

The first and follow sets are as follows:

FIRST SETS:

FIRST(b)	{ b }
FIRST(c)	{ c }
FIRST(a)	{ a }
FIRST(d)	{ d }
FIRST(f)	{ f }
FIRST(e)	{ e }
FIRST(ϵ)	{ ϵ }
FIRST(S)	{ b, c, a, f, e, d }
FIRST(A)	{ a, f, e, ϵ }
FIRST(E)	{ e, ϵ }
FIRST(C)	{ c, ϵ }

FIRST(D)	{ d, ϵ }
----------	-------------------

FOLLOW SETS:

FOLLOW(S)	{ EOF }
FOLLOW(A)	{ d, b, EOF }
FOLLOW(E)	{ d, b, EOF }
FOLLOW(C)	{ EOF }
FOLLOW(D)	{ c }

To prove that the given grammar has a predictive recursive descent parser, it should satisfy the following conditions

1. If $A \rightarrow \alpha$ and $A \rightarrow \beta$ are two grammar rules, then $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
2. If $\epsilon \in FIRST(A)$ then $FIRST(A) \cap FOLLOW(A) = \emptyset$

Condition 1: If a non-terminal symbol appears on the left-hand side of only one rule, condition 1 is immediately satisfied. We consider non-terminals that have more than one rules. These non-terminals are S, A, B, C, D.

a) Non-terminal **S**

$S \rightarrow AbC$

$S \rightarrow DcA$

$FIRST(AbC) = \{a, e, f, b\}$

$FIRST(DcA) = \{d, c\}$

$FIRST(AbC) \cap FIRST(DcA) = \emptyset$

Condition 1 is satisfied for S

b) Non-terminal **A**

$A \rightarrow aAd$

$A \rightarrow E$

$A \rightarrow f$

$FIRST(aAd) = \{a\}$

$FIRST(E) = \{e, \epsilon\}$

$FIRST(f) = \{f\}$

$FIRST(aAd) \cap FIRST(E) = \emptyset$

$$\text{FIRST}(f) \cap \text{FIRST}(E) = \emptyset$$

$$\text{FIRST}(f) \cap \text{FIRST}(aAd) = \emptyset$$

Condition 1 is satisfied for A.

c) Non-terminal **E**

$$E \rightarrow e$$

$$E \rightarrow \epsilon$$

$$\text{FIRST}(e) = \{e\}$$

$$\text{FIRST}(\epsilon) = \{\epsilon\}$$

$$\text{FIRST}(e) \cap \text{FIRST}(\epsilon) = \emptyset$$

Condition 1 is satisfied for E

a) Non-terminal **C**

$$C \rightarrow c$$

$$C \rightarrow \epsilon$$

$$\text{FIRST}(c) = \{c\}$$

$$\text{FIRST}(\epsilon) = \{\epsilon\}$$

$$\text{FIRST}(c) \cap \text{FIRST}(\epsilon) = \emptyset$$

Condition 1 is satisfied for C

b) Non-terminal **D**

$$D \rightarrow dD$$

$$D \rightarrow \epsilon$$

$$\text{FIRST}(dD) = \{d\}$$

$$\text{FIRST}(\epsilon) = \{\epsilon\}$$

$$\text{FIRST}(d) \cap \text{FIRST}(\epsilon) = \emptyset$$

Condition 1 is satisfied for D

We have shown that the grammar satisfies the first condition for predictive parsing. Now, we consider the second condition.

Condition 2: If $\epsilon \in \text{FIRST}(A)$ then $\text{FIRST}(A) \cap \text{FOLLOW}(A) = \emptyset$

For every non-terminal A such that $\epsilon \in \text{FIRST}(A)$, we need to show that $\text{FIRST}(A) \cap \text{FOLLOW}(A) = \emptyset$. The non-terminals that have ϵ in their FIRST sets are A, E, C, and D.

$\text{FIRST}(A) \cap \text{FOLLOW}(A) = \{a, f, e, \epsilon\} \cap \{d, b, \text{EOF}\} = \emptyset$

$\text{FIRST}(E) \cap \text{FOLLOW}(E) = \{e, \epsilon\} \cap \{d, b, \text{EOF}\} = \emptyset$

$\text{FIRST}(C) \cap \text{FOLLOW}(C) = \{c, \epsilon\} \cap \{\text{EOF}\} = \emptyset$

$\text{FIRST}(D) \cap \text{FOLLOW}(D) = \{d, \epsilon\} \cap \{c\} = \emptyset$

As highlighted above Condition 2 is satisfied by Non-terminals A, E, C and D. Therefore, this grammar satisfies condition 2. Given that the grammar satisfies both conditions for the existence of a predictive recursive descent parser, the grammar has a predictive recursive descent parser.

3.2. Write `parse_S()`, `parse_A()`. Your parser 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 .

Solution 3.2

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

```
void parse_S()
{
    // S -> A b C
    // S -> D c A
    // FIRST(A b C) = { a, b, f, e }
    // FIRST(D c A) = { d, c }

    Token t = lexer.peek();
    if ((t.type == a_type) || (t.type == b_type) ||
        (t.type == f_type) || (t.type == e_type)) // S -> A b C
    {
        parse_A();
        lexer.expect(b_type);
        parse_C();
    }
    else if (t.type == d_type || t.type == c_type) // S -> D c A
    {
        parse_D();
        lexer.expect(c_type);
        parse_A();
    }
    else
        syntax_error();
}
```

```

void parse_A()
{
    // A -> a A d
    // A -> E
    // A -> f
    FIRST(a A d) = { a }
    FIRST(E) = { e ,  $\epsilon$  }
    FIRST(f) = { f }
    FOLLOW(A) = { d, b, EOF }

    Token t= lexer.peek();
    if(t.type == a_type)          // A -> a A d
    {
        lexer.expect(a-type);
        parse_A();
        lexer.expect(a-type);
    }
    else if(t.type == e_type)     // A -> E
    {
        parse_E();
    }
    else if(t.type == f_type)     // A -> f
    {
        lexer.expect(f-type);
    }
    else if( (t.type == d)||      // if token in FOLLOW(A) parse rule that
              (t.type == b)||    // generates epsilon: A -> E
              (t.type == EOF) )
    {
        Parse_E();
    }
    else
        syntax_error();
}

```

3.3. Give a full execution trace for your parser from part 3.2 above on input a a d d .

Solution 3.3

```
parse_input()
  parse_S()
    peek()    // next token is a
    parse_A()
      peek()           // next token is a
      expect(a-type)   // a consumed
      parse_A()
        expect(a-type) // a consumed
        parse_A()
          peek()       // next token is d
          parse_E()
            peek()     // next token is d
            return;    // E -> epsilon
          expect(d-type) // d is consumed
        expect(d-type) // d is consumed
      expect(b-type)
      if(t.type != b_type) // true because t.type = EOF
        syntax_error()
```

4. We say that a symbol of a grammar is useless if it does not appear in the derivation of a strings of terminals. Formally, *A is useful* if there exists a derivation $S \Rightarrow^* x A y \Rightarrow^* w$ 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 nonterminal. Also, a useless symbol can be a terminal or a non-terminal.

Consider the grammar

$$S \rightarrow A b C \mid D c A$$
$$A \rightarrow a C$$
$$E \rightarrow e D \mid \epsilon$$
$$C \rightarrow c A$$
$$D \rightarrow d D \mid E$$

Which are the useful symbols of this grammar? For every useful symbol, give a derivation that shows that the symbol is useful.

Answer

Useful symbols are those symbols that appear in derivations of strings of terminals (including the empty string). In the given grammar, there is no derivation starting from S that results in a string of terminals. It follows that S is useless and all symbols are useless.

To see why S cannot generate any string of terminals, consider A. A derivation starting from A will be

$$A \Rightarrow a C \Rightarrow a c A \Rightarrow a c a C \Rightarrow a c a c A \Rightarrow \dots$$

and so on. At no point, can we get a string of terminals starting from A. It follows that A cannot be part of a derivation of a string of terminals. Now, if we consider the two rules for S, we can see that A appears in both of them, so those rules cannot be part of a derivation that generates a string of terminals!

In summary, there is no derivation starting from S that generates a string of terminals, so all symbols are useless.

5. Consider the following operator grammar:

$$\begin{aligned} E &\rightarrow E \& E \\ E &\rightarrow E ! E \\ E &\rightarrow E \wedge E \\ E &\rightarrow \sim E \\ E &\rightarrow (E) \mid id \end{aligned}$$

& is a right associative binary operator. ! and ^ are left associative binary and have the same precedence. & has higher precedence than ! and ^. ~ is a unary operator and has lower precedence than & and higher precedence than ! and ^.

5.1. Draw the precedence table for this grammar:

Solution:

Operator &

- & <• & because & is right associative binary operator
- & •> ! because & has higher precedence and both are binary operator
- & •> ^ because & has higher precedence and both are binary operator
- & <• ~ because ~ is unary operator

Operator !

- ! <• & because & has higher precedence and both are binary operator
- ! •> ! because ! is left associative binary operator
- ! •> ^ because both are left associative binary operators at the same level

- 人。


























































Operator ^

- 人

Operator ~

- 人。

The final table is the following

	&	!	^	~	()	ID	\$
&								
!								
^								
~								
(							
)								
ID								
\$								

5.2. Show step by step how $\sim id \& id ! id ^ id$ is parsed.

Note. Refer to the solution of homework from last semester for the expected format of your answers.

Solution:

Stack	Input	Action	Justification
\$	$\sim id \& id ! id ^ id \$$	Shift	$\$ < \bullet \sim$
\sim	$id \& id ! id ^ id \$$	Shift	$\sim < \bullet id$
$\sim id$	$\& id ! id ^ id \$$	Reduce $E \rightarrow id$	$id \bullet > \&$
$\sim E$	$\& id ! id ^ id \$$	Shift	$\sim < \bullet \&$
$\sim E \&$	$id ! id ^ id \$$	Shift	$\& < \bullet id$
$\sim E \& id$	$! id ^ id \$$	Reduce $E \rightarrow id$	$id \bullet > !$
$\sim E \& E$	$! id ^ id \$$	Reduce $E \rightarrow E \& E$	$\& \bullet > !$
$\sim E$	$! id ^ id \$$	Reduce $E \rightarrow \sim E$	$\sim \bullet > !$
E	$! id ^ id \$$	Shift	$\$ < \bullet !$
$E !$	$id ^ id \$$	Shift	$! < \bullet id$
$E ! id$	$^ id \$$	Reduce $E \rightarrow id$	$id \bullet > ^$
$E ! E$	$^ id \$$	Reduce $E \rightarrow E ! E$	$! \bullet > ^$
E	$^ id \$$	Shift	$\$ < \bullet ^$
$E ^$	$id \$$	Shift	$^ < \bullet id$
$E ^ id$	$\$$	Reduce $E \rightarrow id$	$id \bullet > \$$
$E ^ E$	$\$$	Reduce $E \rightarrow E ^ E$	$^ \bullet > \$$
E	$\$$	return	$\$$ on stack and $\$$ on input

The derivation we obtain is the following (reading the reductions backward)

E => E ^ E
 => E ^ id
 => E ! E ^ id
 => E ! id ^ id
 => ~ E ! id ^ id
 => ~ E & E ! id ^ id
 => ~ E & id ! id ^ id
 => ~ id & id ! id ^ id

Notice that this is a rightmost derivation.

If we want to parenthesize the original expression according to this derivation, we get

$((\sim (id \& id)) ! id) ^ id)$

which is what we expect. & has higher precedence than ~ and ! so id & id will be grouped together. ~ has higher precedence than ! so (~ (id & id)) will be grouped together. This leaves ! and ^ which are both at the same level and both are left associative so we group from left to write.