CSE340 SPRING 2021 - HOMEWORK 2

Due: Wednesday February 17, 2021 by 11:59 PM on Canvas

All submissions should be typed, no exceptions.

1. Consider the grammar.

S → C A B b (1)

A → a A (2)

A → C (3)

B → d B e (4)

B → A B C ! (5)

B→ & (6)

C → C a (7)

C → ε (8)

where S, A, B, and C are the non-terminals, S is the start symbol, and a, b , d , e , ‘&‘ and ‘!’ are the terminals.

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

FIRST(ε) = {ε}

FIRST(a) = {a}

FIRST(b) = {b}

FIRST(d) = {d}

FIRST(e) = {e}

FIRST(&) = {&}

FIRST(!) = {!}

FIRST(S) = {}

FIRST(A) = {}

FIRST(B) = {}

FIRST(C) = {}

* + 1. 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(a) = {a}

FIRST(b) = {b}

FIRST(d) = {d}

FIRST(e) = {e}

FIRST(&) = {&}

FIRST(!) = {!}

FIRST(S) = {}

FIRST(A) = {a}

FIRST(B) = {d, &}

FIRST(C) = {ε}

* + 1. Show the final result of the calculation of the FIRST sets

FIRST(ε) = {ε}

FIRST(a) = {a}

FIRST(b) = {b}

FIRST(d) = {d}

FIRST(e) = {e}

FIRST(&) = {&}

FIRST(!) = {!}

FIRST(S) = {a, d, &}

FIRST(A) = {a, ε}

FIRST(B) = {d, &, a}

FIRST(C) = {ε, a}

* 1. **FOLLOW sets**. Do the following
     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) = {$}

FOLLOW(A) = {d, &, a}

FOLLOW(B) = {b, e, a, !}

FOLLOW(C) = {a, d, &, !}

* + 1. 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) = {$}

FOLLOW(A) = {d, &, a}

FOLLOW(B) = {b, e, a, !}

FOLLOW(C) = {a, d, &, !}

* + 1. Show the final result of the calculation of the FOLLOW sets

FOLLOW(S) = {$}

FOLLOW(A) = {d, &, a}

FOLLOW(B) = {b, e, a, !}

FOLLOW(C) = {a, d, &, !}

1. Consider the grammar

S → B A | C D

A → c A b | e

B → D C | C

C → c B c | 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. 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.

The FIRST sets are as follows:

FIRST(ε) = {ε}

FIRST(b) = {b}

FIRST(c) = {c}

FIRST(d) = {d}

FIRST(e) = {e}

FIRST(S) = {c, d, ε, e}

FIRST(A) = {c, e}

FIRST(B) = {d, c, ε}

FIRST(C) = {c, ε}

FIRST(D) = {d, ε}

The FOLLOW sets are as follows:

FOLLOW(S) = {$}

FOLLOW(A) = {b, $}

FOLLOW(B) = {c, e}

FOLLOW(C) = {d, $, c, e}

FOLLOW(D) = {c, $, e}

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

1. If A →  and A →  are two grammar rules, then FIRST() ∩ FIRST() = ∅
2. If ε ∈ FIRST(A), then FIRST(A) ∩ FOLLOW(A) = ∅

**Condition 1:**

If the 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 S, A, B, C and D.

For non-terminal S:

FIRST(BA) = {d, c, e}

FIRST(CD) = {c, d, ε}

FIRST(BA) ∩ FIRST(CD) ≠ ∅

Hence, condition 1 is not satisfied for S

For non-terminal A:

FIRST(cAb) = {b}

FIRST(e) = {e}

FIRST(cAb) ∩ FIRST(e) = ∅

Hence, condition 1 is satisfied for A.

For non-terminal B:

FIRST(DC) = {d, c, ε}

FIRST(C) = {c, ε}

FIRST(DC) ∩ FIRST(C) ≠ ∅

Hence, condition 1 is not satisfied for B

For non-terminal C:

FIRST(cBc) = {c}

FIRST(ε) = {ε}

FIRST(cBc) ∩ FIRST(ε) = ∅

Hence, condition 1 is satisfied for C

For non-terminal D:

FIRST(d) = {d}

FIRST(ε) = {ε}

FIRST(d) ∩ FIRST(ε) = ∅

Hence, condition 1 is satisfied for D.

Condition 1 is not satisfied for non-terminals: S and B

**Condition 2:**

For every non-terminal X such that ε ∈ FIRST(X), we need to show that FIRST(X) ∩ FOLLOW(X) = ∅. The non-terminals that have ε in their first sets are S, A, B, C and D.

FIRST(S) ∩ FOLLOW(S) = {c, d, ε, e}∩ {$} = ∅

FIRST(A) ∩ FOLLOW(A) = {c, e}∩ {b, $} = ∅

FIRST(B) ∩ FOLLOW(B) = {d, c, ε} ∩{c, e} = {c} ≠ ∅

FIRST(C) ∩ FOLLOW(C) = {c, ε} ∩{d, $, c, e} = {c} ≠ ∅

FIRST(D) ∩ FOLLOW(D) = {d, ε} ∩ {c, $, e} = ∅

Condition 2 is not satisfied for non-terminals: B and D

1. Consider the grammar

S → C b A | f B C

A → b B A | C D

B → a B | ε

C → c | E

D → d D | ε

E → e E | ε

The FIRST sets are as follows:

FIRST(ε) = {ε}

FIRST(a) = {a}

FIRST(b) = {b}

FIRST(c) = {c}

FIRST(d) = {d}

FIRST(e) = {e}

FIRST(f) = {f}

FIRST(S) = {f, c, e, b}

FIRST(A) = {b, c, e, d, ε}

FIRST(B) = {a, ε}

FIRST(C) = {c, e, ε}

FIRST(D) = {d, ε}

FIRST(E) = {e, ε}

The FOLLOW sets are as follows:

FOLLOW(S) = {$}

FOLLOW(A) = {$}

FOLLOW(B) = {c, e, b, d, $}

FOLLOW(C) = {b, d, $}

FOLLOW(D) = {$}

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

* 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.

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

1. If A →  and A →  are two grammar rules, then FIRST() ∩ FIRST() = ∅
2. If ε ∈ FIRST(A), then FIRST(A) ∩ FOLLOW(A) = ∅

**Condition 1:**

For non-terminal S:

FIRST(CbA) = {c, e, b}

FIRST(fBC) = {f}

FIRST(CbA) ∩ FIRST(fBC) = ∅

Hence, condition 1 is satisfied for S

For non-terminal A:

FIRST(bBA) = {b}

FIRST(CD) = {c, e, d, ε}

FIRST(bBA) ∩ FIRST(CD) = ∅

Hence, condition 1 is satisfied for A

For non-terminal B:

FIRST(aB) = {a}

FIRST(ε) = {ε}

FIRST(aB) ∩ FIRST(ε) = ∅

Hence, condition 1 is satisfied for B

For non-terminal C:

FIRST(c) = {c}

FIRST(E) = {e, ε}

FIRST(c) ∩ FIRST(E) = ∅

Hence, condition 1 is satisfied for C

For non-terminal D:

FIRST(dD) = {d}

FIRST(ε) = {ε}

FIRST(dD) ∩ FIRST(ε) = ∅

Hence, condition 1 is satisfied for D

For non-terminal E:

FIRST(eE) = {e}

FIRST(ε) = {ε}

FIRST(eE) ∩ FIRST(ε) = ∅

Hence, condition 1 is satisfied for E

Hence, condition 1 is satisfied for all non-terminals

**Condition 2:**

For every non-terminal X such that ε ∈ FIRST(X), we need to show that FIRST(X) ∩ FOLLOW(X) = ∅. The non-terminals that have ε in their first sets are S, A, B, C and D.

FIRST(S) ∩ FOLLOW(S) = {f, c, e, b} ∩{$} = ∅

FIRST(A) ∩ FOLLOW(A) = {b, c, e, d, ε} ∩{$} = ∅

FIRST(B) ∩ FOLLOW(B) = {a, ε} ∩{c, e, b, d, $} = ∅

FIRST(C) ∩ FOLLOW(C) = {c, e, ε} ∩{b, d, $} = ∅

FIRST(D) ∩ FOLLOW(D) = {d, ε} ∩{$} = ∅

FIRST(E) ∩ FOLLOW(E) = {e, ε} ∩{b, d, $} = ∅

Hence, condition 2 is satisfied for all non-terminals

* 1. Write parse\_input()

|  |
| --- |
| void **parse\_input**() {  parse\_S();  lexer.expect(EOF); } |

* 1. Write parse\_S()

|  |
| --- |
| void **parse\_S**() { // S -> C b A  // S -> f B C  // FIRST(C b A) = { c, e, b}  // FIRST(f B C) = { f }  Token t = lexer.peek(1);  if ((t.token\_type == c\_type)||(t.token\_type == e\_type) ||  (t.token\_type == b\_type)) // S -> C b A  {  parse\_C();  lexer.expect(b\_type);  parse\_A();   }  else if (t.token\_type == f\_type) // S -> f B C  {  lexer.expect(f-type);  parse\_B();  parse\_C();  }  else   syntax\_error(); } |

* 1. Write parse\_C().

|  |
| --- |
| void **parse\_C**() { *// C -> c*  *// C -> E*  *// FIRST(c) = { c }*  *// FIRST(E) = { e, ε }*  *// FOLLOW(C) = { b, d, EOF}*  Token t = lexer.peek();  if (t.token\_type == c\_type) *// C -> c*  {  lexer.expect(c\_type);   }  else if (t.token\_type == e\_type) *// C -> E*  {  parse\_E();  }  else if ((t.token\_type == b\_type) || *// Since, C generates epsilon, if*  (t.token\_type == d\_type))|| *// token in FOLLOW(C) parse rule*  (t.token\_type == EOF)) *// that generates epsilon: C -> E*  {  parse\_E();  }  else  syntax\_error(); } |

* 1. Give a **full execution** trace for your parser from part 3.2 above on input c b d c

Note that this input will necessitate calling the parse\_C() function. Your trace should show the correct sequence of calls including the parse\_C() function and other functions needed in the trace even if you are not asked to write them in parts 3.2, 3.3 and 3.4 above.

|  |
| --- |
| parse\_input()  parse\_S()  peek() *// next token is c*  parse\_C()  peek() *// next token is c*  expect(c-type) *// c consumed*  expect(b-type) *// b consumed*  parse\_A()  peek() *// next token is d*  parse\_C()  peek() *// next token is d*  parse\_E()  peek() *// next token is d*  return; *// E -> epsilon*  parse\_D()  peek() *// next token is d*  expect(d-type) *// d consumed*  parse\_D()   peek() *// next token is c but EOF or d expected*  syntax\_error() *// Hence, syntax\_error* |

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.

1. 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 uselss 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 C | 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 D | F g

F → f F| a

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.

S, A, B, C, D and F are not useless because they appears in the following derivation of a b d f a g b a g c

S ⇒ ABC

⇒ a B B C

⇒ a b D b D C

⇒ a b d D b F g C

⇒ a b d F g b a g c

⇒ a b d f F g b a g c

⇒ a b d f a g b a g c

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. e is useless because there is no derivation of a string of terminals starting from S in which e appears.