PREDICTIVE PARSING
with
FIRST AND FOLLOW SETS

CSE 340 FALL 2021

Rida Bazzi

PREDICTIVE PARSING with FIRST AND FOLLOW SETS

CSE 340 FALL 2021

Rida Bazzi

Contents

1.	Motivating Example	pages 3 - 12
2.	FIRST and FOLLOW	page 13
3.	Definitions of FIRST and FOLLOW	page 14
4.	Note about FIRST and FOLLOW	page 15
5.	Conditions for the existence of a predictive parser	pages 16 – 18
6.	General form of a predictive parser with example	pages 19 – 26
7.	rules for calculating FIRST sets	pages 27 - 28
8.	Rules for calculating FOLLOW sets	pages 29 - 30
9.	Calculating FIRST sets example	pages 31 – 40
10.	Calculating FOLLOW sets example	pages 41 - 45

Motivating Example

Consider the grammar

S -> A B
A -> a C | D
C -> c C | ε
D -> E F
E -> e E | ε
F -> f F | ε
B -> b B | ε

I will show how to write a parser for this grammar and explain the choices I make as I write the parser.

This example will be an introduction to a general approach to parsing that is called predictive parsing

General rules for writing the Parse_X() functions where X is a non-terminal

For each non-terminal we need to write a parsing function (hence the heading of this page). The parsing functions we will write satisfy the following properties:

- When parse_X() is called, it consumes part of the input that corresponds to X.
- When parse_X() returns, it would not have consumed any input that does not correspond to X. Any tokens that are not part of X should be obtained through peek() and a token should be consumed by parse_X() only if it is part of X.
- Every successful execution path of parse_X() must correspond to consuming one of the righthand sides of the rules for X (an execution path is successful if it does not generate syntax_error()).

If these general rules are not clear, they should become clearer as we go through the example

Writing the parsing functions for the grammar: parse_input()

I repeat the grammar for convenience

S -> A B
A -> a C | D
C -> c C | ε
D -> E F
E -> e E | ε
F -> f F | ε
B -> b B | ε

We start by writing parse_input(). Since the start symbol is S*, the input must consist of S with nothing after it. So, the function parse_input() will look as follows

```
parse_input()
{
      parse_S();
      expect(EOF);
}
```

Let us examine this function. First parse_S() is called. Since parse_S() should be written so that it consumes the part of the input that corresponds to S and nothing else (see previous page), if parse_S() returns successfully, we know that parse_S() must have consumed "an S". So, it remains for parse_input() to determine that there is nothing left in the input which would mean that the input consists of S followed by nothing and it is valid input.

To check that there is nothing after parse_S(), parse_input() calls expect(EOF). Recall that expect(expected_type) consists of a getToken() followed by a test that the type of the returned token is equal to expected_type. If it is equal to the expected type, expect() simply returns the token obtained by getToken(), otherwise it calls syntax_error(), which exits the program. If expect(EOF) succeeds, this means that there are no tokens remaining in the input after S and parse_input() succeeds. If there are more tokens after S in the input, expect(EOF) will fail and results in syntax_error().

Even though, this function is relatively straightforward, the explanation should help you in getting a better understanding. Now, we move to parse_S() on the next page.

* By convention, unless otherwise specified, the start symbol is the left-hand side of the first rule. See second set of notes for this class about this convention and other conventions for context-free grammars.

Writing the parsing functions for the grammar: parse_S()

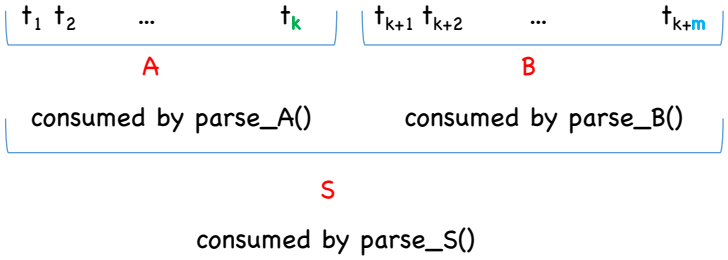
I repeat the grammar for convenience

S -> A B
A -> a C | D
C -> c C | ε
D -> E F
E -> e E | ε
F -> f F | ε
B -> b B | ε

There is only one rule for S in the grammar, namely: S - > A B. We understand this to mean that an S consists of an A followed by a B. So, in order to parse_S() successfully, we need to determine that the input can be broken into two parts one after the other. The first part must correspond to A and the second part must correspond to B. Luckily, we have the functions that can do this for us: parse_A() and parse_B() which should be written to consume parts of the input that correspond to A and B respectively. So, parse_S() will be

```
parse_S()
{
        parse_A();
        parse_B();
}
```

Even though this code might be clear to you, it is worth explaining what parse_S() is doing. When parse_A() is called, it consumes part of the input that corresponds to A (if there is no syntax error), then when parse_B() is called, it starts parsing from the point parse_A() left off, which means it starts at the next token after the "A" that was parsed by parse_A(). When parse_B() returns, it would have consumed input that corresponds to B. So, the two calls in a sequence have the net result of consuming input corresponding to A followed by input corresponding to B, which means the whole thing corresponds to S as illustrated below.



In the illustration there are k+m tokens. The first k tokens correspond to A and are consumed by parse_A(). The next m tokens correspond to B and are consumed by parse_B(). The total k+m tokens is consumed by parse_S() by first calling parse_A() and then calling parse_B().

Writing the parsing functions for the grammar: parse_A()

I repeat the grammar for convenience

S -> A B
A -> a C | D
C -> c C | ε
D -> E F
E -> e E | ε
F -> f F | ε
B -> b B | ε

parse_A() is more involved. There are two rules for A each giving a different "form" of what A "looks like". In order to determine according to which rule we need to parse A, we need to peek() to guide the parsing.

If you examine the rules for A, you will notice that a C can only generate sequences of tokens starting with a, whereas D can generate sequences of tokens starting with e or f but cannot generate sequences starting with a. To see why D can only generate sequences of tokens starting with e or f, consider the possibilities

D => E F => e E F (in the second derivation step we are using E -> e E)

D => E F => F => f F (in the second derivation step we are using E -> ε)

If you think about it, you should be able to convince yourself that these are the only possibilities for D (the possibility of D generating epsilon will be considered later).

So, parse_A() will look as shown on the next page

Writing the parsing functions for the grammar: parse_A()

I repeat the grammar for convenience

S -> A B
A -> a C | D
C -> c C | ε
D -> E F
E -> e E | ε
F -> f F | ε
B -> b B | ε

parse_A() will look as follows

```
parse_A()
{
    t = lexer.peek(1);                  // this peeks at the next token without
                                        // consuming it

    if (t.type == a-type)               // in this case we should parse according
    {                                   // to A -> a C
            expect(a-type);             consumes a
            parse_C();                  consumes part of the input
                                        that corresponds to C

    } else if  (  (t.type == e-type) |  // in this case we should parse
                  (t.type == f-type) )  // according to A -> D
    {             parse_D();            consumes part of the input
                                        that corresponds to D

    } else
        ...                             
}
```

It should be clear that the code is not complete with the "..." at the end. I will finish it on the next page. But at least for the part that is shown, it should be clear what is going on:

parse_A() starts by peeking at the next token. If the next token is a, then parse_A() should parse according to the A -> a C rule. To parse according to the rule A -> a C, we need to first consume the a and then parse the C. This is highlighted in the code above. To consume the a, we get a token and make sure that it is a, which is achieved with expect(a-type). To parse C, we simply call parse_C(). In the code above, for this particular example, calling expect(a-type) is redundant because we already checked for a before entering the body of the if, so we could have easily called getToken() instead of expect(). Nonetheless, this is the systematic way to consume a specific token (consider what you would write if the rule for A was A -> a C b and you want to consume the b after parsing the C).

We continue on the next page ....

Writing the parsing functions for the grammar: parse_A()

Now, we consider what happens if the next token is not a, e or f. We might think that it would be safe to assume in this case that A must be epsilon, which means that we should call parse_D() in the part occupied by "..." on the previous page. This is indeed doable and if there is syntax error it would be detected in some other parsing function.

Nonetheless, since we already have the token, we can determine if A should be epsilon. In fact, if A is indeed epsilon, the token we get must come from B, or, if B itself is also epsilon, then we should get EOF. This is made clearer by the following derivations:

1. S => A B => A b B
2. S EOF => A B EOF => A EOF

In the first derivation, if A should be parsed as epsilon (A => D => E F => E => ε), the token that we get when we attempt to parse_A() will be b. In the second case when A and B derive epsilon, the token we get when we attempt to parse_A() is EOF.

The discussion above is captured in the code below

```
parse_A()
{
    t = lexer.peek(1);              // this peeks at the next token without
                                    // consuming it

    if (t.type == a-type)           // in this case we should parse according
    {                               // to A -> a C
            expect(a-type);         // consume a
            parse_C();              // consumes part of the input
                                    // that corresponds to C

    } else if  (  (t.type == e-type) |   // in this case we should parse
                  (t.type == f-type) )   // according to A -> D
    {             parse_D();        // consumes part of the input
                                    // that corresponds to D
    } else if (   (t.type = b-type) |    // if the token is not part of A but part of
                  (t.type = EOF) )  // FOLLOW(A) parse according to RHS that can
                                    // derive epsilon which is A -> D
    {             parse_D();        // parse D because this is the only
                                    // way to get epsilon
    } else
            syntax_error();
}
```

It is important to note that when the peek() returns b or EOF, we cannot simply call return instead of calling parse_D(). If we just return without calling parse_D(), then we are effectively saying that A => ε, which is not correct according to the grammar. The correct derivation is A => D => ε

The remaining functions

I repeat the grammar for convenience

S -> A B
A -> a C | D
C -> c C | ε
D -> E F
E -> e E | ε
F -> f F | ε
B -> b B | ε

The remaining functions are given with some explanations after each

```
parse_C()
{
        t = lexer.peek(1);                  // this peeks at the next token without
                                            // consuming it

        if (t.type == c-type)               // in this case we should parse according
        {                                   // to C -> c C
                    expect(c-type);                    consume c
                    parse_C();                         consumes part of the input
                                                       that corresponds to C

        } else if ( (t.type == b-type) | (t.type == EOF) )
        {
                    return;                            C -> ε

        } else
                    syntax_error();

}
```

The parse_C() function is similar to parse_A(). It is interesting to note that if C is epsilon (C generates epsilon), getToken() would return b or EOF when called in parse_C().

The remaining functions

I repeat the grammar for convenience (are you tired of this yet?)

S -> A B
A -> a C | D
C -> c C | ε
D -> E F
E -> e E | ε
F -> f F | ε
B -> b B | ε

parse_D() and parse_E() are given below with some comments after each one

```
parse_D()
{
        parse_E();
        parse_F();
}
```

Notice that since D has only one rule, we just parse according to that one rule.

```
parse_E()
{
        t = lexer.peek(1);
        if (t.type == e-type)
        {
                expect(e-type);                 consume e
                parse_E();                      parse E
        } else if (   (t.type == f-type) |      // f,
                      (t.type == b-type) |      // b and
                      (t.type == EOF) )         // EOF follow E
        {
                return;                         E is epsilon
        } else
                syntax_error();

}
```

Here we have an interesting new case. It should be clear why f can come after E when E is epsilon. It might not be as clear why b and EOF can come after E. Here is an explanation

S => A B => D B => E F B => E B

Notice how when F is epsilon, B is after E in the derivation which explains why b and EOF can come after E.

The remaining functions

I repeat the grammar for convenience

S -> A B
A -> a C | D
C -> c C | ε
D -> E F
E -> e E | ε
F -> f F | ε
B -> b B | ε

Finally, we write parse_F() and parse_B()

```
parse_F()
{
        t = lexer.peek(1);
        if (t.type == f-type)
        {
                expect(f-type);
                parse_F();
        } else if ( (t.type == b-type) | (t.type == EOF) )
        {
                return;
        } else
                syntax_error();
}

parse_B()
{
        t = lexer.peek(1);
        if (t.type == b-type)
        {
                expect(b-type);
                parse_B();
        } else if (t.type == EOF)
        {
                return;
        } else
                syntax_error();
}
```

FIRST and FOLLOW

- FIRST sets are sets that contains symbols from T ∪ { Ɛ } and are calculated for every terminal and non-terminal symbol of the grammar
- FOLLOW sets are sets that contain symbols from T ∪ { $ } and, for our purposes in this class, are calculated only for non-terminals

FIRST and FOLLOW sets are essential for writing parsers, even other kinds of parsers like bottom-up parsers.

In these notes, I will define FIRST and FOLLOW, show how they are used to write predictive parsers, and give the algorithms for calculating them.

I start by giving an example of FIRST and FOLLOW sets for a given grammar without showing how they are calculated.

GRAMMAR RULES		FIRST SETS	FOLLOW SETS
S -> A B C	(1)	FIRST(Ɛ) = { Ɛ }	FOLLOW(S) = { $ }
A -> B C d D E	(2)	FIRST(b) = { b }	FOLLOW(A) = { b , e, c , $ }
B -> b B	(3)	FIRST(c) = { c }	FOLLOW(B) = { c , d , $}
B -> D E	(4)	FIRST(d) = { d }	FOLLOW(C) = { d , $ }
D -> Ɛ	(5)	FIRST(e) = { e }	FOLLOW(D) = { e, b, c, $, d }
E -> Ɛ	(6)	FIRST(S) = { b , e, c , d }	FOLLOW(E) = {b, e, c, $, d }
E -> e	(7)	FIRST(A) = { b, e, c , d }	
C -> c C	(8)	FIRST(B) = { b , e , Ɛ }	
C -> Ɛ	(9)	FIRST(C) = { c , Ɛ }	
		FIRST(D) = { Ɛ }	
		FIRST(E) = { Ɛ , e }	

Definitions of FIRST and FOLLOW

The following are the definitions of FIRST and FOLLOW sets. Later we will see how we can calculate these sets.

FIRST Sets FIRST sets are calculated for terminals or non-terminals.

1. terminal a ∈ FIRST(A) where A is a terminal or non-terminal if and only if for some string w of terminals and non-terminals (possibly empty)

$$A \stackrel{*}{\Rightarrow} a\ w$$

2. $\varepsilon \in$ FIRST(A) if and only if

$$A \stackrel{*}{\Rightarrow} \varepsilon$$

FOLLOW sets We calculate follow sets for non-terminals

a symbol a which is a terminal or $ (to denote end of file or EOF) is in FOLLOW(A) if and only if

$$S\ \$ \stackrel{*}{\Rightarrow} x\ A\ a\ y$$

Remember

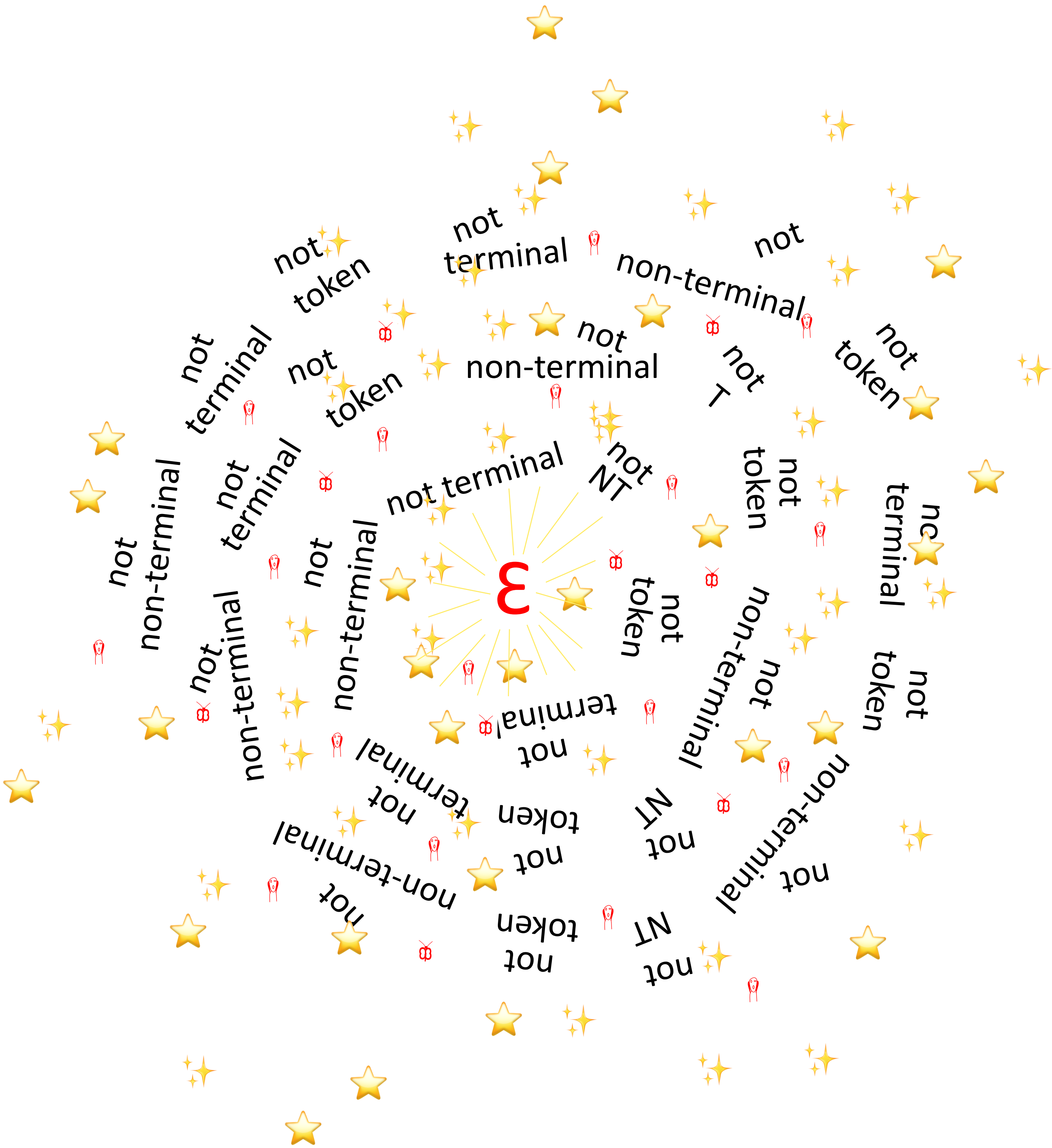
ε is NOT a terminal

ε is NOT a token

getToken() never returns ε

ε cannot be an element of a FOLLOW set EVER

$ cannot be an element of a FIRST set EVER



ε
not terminal
not NT
not token
not terminal
not non-terminal
not terminal
not token
not non-terminal
not T
not non-terminal
not terminal
not token
not terminal
not non-terminal
not non-terminal
not terminal
not token
not terminal
not NT
not token
not non-terminal
not token
not NT
not token
not non-terminal
not token
not terminal
not non-terminal
not token
not terminal
not non-terminal
not token

Conditions for predictive parsing

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

1. For every non-terminal A and any two rules

$$\left.\begin{array}{l} A \to \alpha \\ A \to \beta \end{array}\right\} \Longrightarrow \quad FIRST(\alpha) \cap FIRST(\beta) = \emptyset$$

2. If a non-terminal A derives ε ($A \overset{*}{\Rightarrow} \varepsilon$), then the FIRST and FOLLOW sets of A are disjoint

$$\left.\begin{array}{l} A \overset{*}{\Rightarrow} \varepsilon \end{array}\right\} \Longrightarrow \quad FIRST(A) \cap FOLLOW(A) = \emptyset$$

Example 1

S -> A B | C
A -> a A | B C
B -> b B | ε
C -> c C | D
D -> d

If we calculate the FIRST sets of non-terminals, we get

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

We start with S. There are two rules

S -> A B and S -> C

FIRST(AB) = { a , b , c , d }
FIRST(C) = { c , d }

The intersection is not empty because both sets have c and d in common, so the grammar has no predictive parser

* IMPORTANT NOTE. The conditions for predictive parsing assume that the grammar has no useless symbols.

Conditions for predictive parsing

Example 2

S -> A e B | g C
A -> a A | B C | f A
B -> b B | ε
C -> c C | D | ε
D -> d

If we calculate the FIRST sets of non-terminals, we get

FIRST(S) = { a , b , c , d , f , e, g }
FIRST(A) = { a , b , c , d , f , ε }
FIRST(B) = { b , ε }
FIRST(C) = { c , d , ε }
FIRST(D) = { d }

FOLLOW(S) = { $ }
FOLLOW(A) = { e }
FOLLOW(B) = { c , d , e , $ }
FOLLOW(C) = { e , $ }
FOLLOW(D) = { e , $ }

We start with S. There are two rules

S -> A e B and S -> e C

FIRST(A e B) = { a , b , c , d , f , e }
FIRST(g C) = { g }

The intersection is empty, so condition 1 is satisfied for the rules of S.

We do not need to check the second condition of predictive parsing for S because ε ∉ FIRST(S).

Next, we look at the rules for A. There are three rules for A

A -> a A A -> B C and A -> f A

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

The three sets are disjoint, so condition 1 for predictive parsing is satisfied for A.

We need to check the second condition of predictive parsing for A because ε ∈ FIRST(A). Condition 2 holds because FIRST(A) and FOLLOW(A) have no elements in common (see sets above)

.... example continued on next page

Conditions for predictive parsing

Example 2 continued

S -> A e B | g C
A -> a A | B C | f A
B -> b B | ε
C -> c C | D | ε
D -> d

If we calculate the FIRST sets of non-terminals, we get

FIRST(S) = { a , b , c , d , f , e, g }	FOLLOW(S) = { $ }
FIRST(A) = { a , b , c , d , f , ε }	FOLLOW(A) = { e }
FIRST(B) = { b , ε }	FOLLOW(B) = { c , d , e , $ }
FIRST(C) = { c , d, ε }	FOLLOW(C) = { e , $ }
FIRST(D) = { d }	FOLLOW(D) = { e , $ }

Next, we look at the rules for B. There are two rules for B

B -> b B and B -> ε

FIRST(b B) = { b }
FIRST(ε) = { ε }

The two sets are disjoint, so condition 1 for predictive parsing is satisfied for B.

Since, ε ∈ FIRST(B) , we need to check if FIRST(B) ∩ FOLLOW(B) = ∅

FIRST(B) and FOLLOW(B) are disjoint as can be seen from the sets above, so condition 2 for predictive parsing holds for B

Next, we look at the rules for C. There are three rules for C

C -> c C C -> D and C -> ε

FIRST(c C) = { c }
FIRST(D) = { d }
FIRST(ε) = { ε }

The three sets are disjoint, so condition 1 for predictive parsing is satisfied for C.

We need to check the second condition of predictive parsing for C because ε ∈ FIRST(C). We can see from the sets above that FIRST(C) ∩ FOLLOW(C) = ∅, so condition 2 of predictive parsing holds for C.

Finally, we look at the rules for D. There is only one rule for D and because ε ∉ FIRST(D). so The conditions for predictive parsing are satisfied for D.

So, the conditions for predictive parsing are satisfied for G and G has a predictive parser.

Predictive Recursive Descent Parsers from FIRST And FOLLOW sets

Now that we have introduced the conditions for predictive parsing and looked at two examples for how to apply these conditions for a particular grammar, we are ready to introduce the general form of a predictive parser.

I am going to assume that we have a grammar G and that we have calculated the FIRST sets for all righthand sides of rules and that we have also calculated the FOLLOW sets for all non-terminals

The predictive parser will have one parse function for each non-terminal

The form of the parse function for a non-terminal depends on whether or not the non-terminal can derive the empty string

Parse function if A $\overset{*}{\not\Rightarrow}$ ε

```
parse_A()
{
      // Grammar rules for A are
      //    A -> RHS1
      //    A -> RHS2
      ....
      //    A -> RHSk

      t = lexer.peek(1);          // this peeks at the next token

      if t.token_type ∈ FIRST(RHS1)
            // parse RHS1

      else if t.token_type ∈ FIRST(RHS2)
            // parse RHS2

      ....

      else if t.token_type ∈ FIRST(RHSk)
            // parse RHSk

      else
            syntax_error();

}
```

Parse function if A $\stackrel{*}{\Rightarrow}$ ε

```
parse_A()
{
      // Grammar rules for A are
      //    A -> RHS1
      //    A -> RHS2
      ....
      //    A -> RHSk

      t = lexer.peek(1);          // this peeks at the next token

      if t.token_type ∈ FIRST(RHS1)
            // parse RHS1

      else if t.token_type ∈ FIRST(RHS2)
            // parse RHS2

      ....

      else if t.token_type ∈ FIRST(RHSk)
            // parse RHSk

      else if t.token_type ∈ FOLLOW(A)
            // parse the RHS that can
            // generate ε

      else
            syntax_error();

}
```

Parsing Righthand Sides

To complete the parse function for non-terminals, we need to specify how to parse the righthand sides of rules.

Let A -> A1 A2 A3 … Ak be a grammar rule, where A1, …, Ak are terminals and non-terminals (if the sequence on the RHS is empty, we have the rule A -> ε

for i = 1 to k

if Ai is a terminal, "write"

expect(Ai-type);

else if Ai is a non-terminal, "write"

parse_Ai();

If the righthand side is empty, you can simply write a semicolon.

This is the recipe for writing parse RHS of the form A1 A2 … Ak

Example A -> a A B b

A -> A1 A2 A3 A4

lexer.expect(a-type); } a

parse_A(); } A

parse_B(); } B

lexer.expect(b-type); } b

Example Predictive Parser

S -> A e B | g C
A -> a A | B C | f A
B -> b B | ε
C -> c C | D | ε
D -> d

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

        t = lexer.peek(1);

        if ( (t.token_type == a-type) | (t.token_type == b-type) |
             (t.token_type == c-type) | (t.token_type == d-type) |
             (t.token_type == f-type) | (t.token_type == e-type) )
                                                // FIRST(A e B) = { a , b , c , d , f , e}
        {
                parse_A();                      // A

                expect(e-type);                 // e

                parse_B();                      // B
        }
        else if (t.token_type == g-type)        // FIRST(g C) =  { g }
        {
                expect(g-type);                 // g

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

Example Predictive Parser

```
S -> A e B | g C
A -> a A | B C | f A
B -> b B | ε
C -> c C | D | ε
D -> d


parse_A()
{           // A -> a A | B C | f A
            // FIRST(a A ) = { a }
            // FIRST(B C) =  { b , c , d , ε }
            // A => BC =>* ε and FOLLOW(A) = { e }


            t = lexer.peek(1);

            if (t.token_type == a-type)                  // FIRST(a A ) = { a }
            {
                        expect(a-type);                  // a

                        parse_A();                       // A
            }
            else if (   (t.token_type == b-type)   |     // FIRST(B C) =  { b , c , d , ε }
                        (t.token_type == c-type)    |
                        (t.token_type == d-type)    )
            {
                        parse_B();                       // B
                        parse_C();                       // C
            }
            else if (t.token_type == f-type)             // FIRST(f A  ) = { f }
            {

                        expect(f-type);                  // f

                        parse_A();                       // A
            }
            else if (t.token_type == e-type)             // A derives ε and FOLLOW(A) = { e }
            {
                        parse_B();                       // B C is the rule that derives
                        parse_C();                       // ε because ε ∈ FIRST( B C )
            } else
                        syntax_error();
}
```

Example Predictive Parser

S -> A e B | g C
A -> a A | B C | f A
B -> b B | ε
C -> c C | D | ε
D -> d

```
parse_B()
{           // B -> b B | ε
            // FIRST(b B ) = { b }
            // B -> ε and FOLLOW(B) =  { c , d , e , $ }

            t = lexer.peek(1);

            if (t.token_type == b-type)                  // FIRST(b B ) = { b }
            {
                        expect(b-type);                  // b
                        parse_B();                       // B
            }
            else if (   (t.token_type == c-type)   |     // B derives ε and
                        (t.token_type == d-type)   |     // FOLLOW(B) =  { c , d , e,  $ }
                        (t.token_type == e-type)   |
                        (t.token_type == EOF )     )
            {
                        return;                          // B -> ε
            }
            else
                        syntax_error();
}
```

Example Predictive Parser

```
S -> A e B |  g C
A -> a A | B C | f A
B -> b B | ε
C -> c C | D | ε
D -> d


parse_C()
{           // C -> c C | D |  ε
            // FIRST(c C ) = { c }
            // FIRST(D) = { d }
            // C ->  ε  and FOLLOW(C) =  { e , $ }


            t = lexer.peek(1);

            if (t.token_type == c-type)                 // FIRST(c C ) = { c }
            {
                        expect(c-type);                 // c
                        parse_C();                      // C
            }
            else if (t.token_type == d-type)            // FIRST(D) = { d }
            {
                        parse_D();                      // D
            }
            else if (   (t.token_type == e-type)   |    // C derives ε and
                        (t.token_type == EOF)    )      // FOLLOW(C) =  { e , $ }
            {
                        return;                         // C -> ε
            }
            else
                        syntax_error();

}

parse_D()
{           // D -> d

            // Since there is only one rule, we do not need to peek() to "predict" the
            // righthand side to parse. There is only one righthand side
            expect(d-type);
}
```

Rules for Calculating FIRST sets

We use the following 5 rules for calculating FIRST sets for terminals and non-terminals and epsilon

I. FIRST(Ɛ) = { Ɛ }

II. FIRST(a) = { a } for every terminal a

III. If A -> B α is a grammar rule, where B is a terminal or non-terminal, then

add FIRST(B) – { Ɛ } to FIRST(A)

IV. If A -> A1 A2 … Ak B α is a grammar rule, where B is a terminal or non-terminal and
Ɛ ∈ FIRST(A1) and Ɛ ∈ FIRST(A2) …. and Ɛ ∈ FIRST(Ak), then

add FIRST(B) – { Ɛ } to FIRST(A)

V. If A -> A1 A2 … Ak is a grammar rule and
Ɛ ∈ FIRST(A1) and Ɛ ∈ FIRST(A2) …. and Ɛ ∈ FIRST(Ak), then

add Ɛ to FIRST(A)

Calculating FIRST sets

Start by applying rules I and II to get the initial FIRST sets for terminals and epsilon

Next initialize all FIRST sets for non-terminals to empty

Run the following pseudocode

```
change = true;
while changed
{
    changed = false
    for every grammar rule
            apply FIRST set rules III, IV, and V to the grammar rule
            if any FIRST set changes set changed = true
}
```

Rules for Calculating FOLLOW sets

We use the following 5 rules for calculating FOLLOW sets for non-terminals

I. add $ to FOLLOW(S)

II. If A -> α B is a grammar rule, and B is a non-terminal,

add FOLLOW(A) to FOLLOW(B)

III. If A -> α B A1 A2 … Ak is a grammar rule, and B is a non-terminal, and
ε ∈ FIRST(A1) and ε ∈ FIRST(A2) …. and ε ∈ FIRST(Ak), then

add FOLLOW(A) to FOLLOW(B)

IV. If A -> α B A1 A2 … Ak is a grammar rule, where B is non-terminal

add FIRST(A1) – { ε } to FOLLOW(B)

V. If A -> α B A1 A2 …Ai Ai+1 Ak is a grammar rule, where B is non-terminal and
ε ∈ FIRST(A1) and ε ∈ FIRST(A2) …. and ε ∈ FIRST(Ai), then

add FIRST(Ai+1) – { ε } to FOLLOW(B)

Calculating FOLLOW sets

Start by applying rules I to add $ to FOLLOW(S)

Next initialize all FOLLOW sets for non-terminals to empty

Run the following pseudocode

```
// first pass apply rules IV and V to all grammar rules
for every grammar rule
    apply FOLLOW set rules  IV, and V to the grammar rule
```

```
// multiple passes until there is no change
// apply FOLLOW rules II and III to all grammar rules
change = true;
while changed
{
    changed = false
    for every grammar rule
            apply FOLLOW set rules II and III to the grammar rule
            if any FOLLOW set changes set changed = true
}
```

Calculating FIRST sets example

S -> A B C	(1)	FIRST(ε) = { ε }
A -> B C d D E	(2)	FIRST(b) = { b }
B -> b B	(3)	FIRST(c) = { c }
B -> D E	(4)	FIRST(d) = { d }
D -> ε	(5)	FIRST(e) = { e }
E -> ε	(6)	FIRST(S) = { }
E -> e	(7)	FIRST(A) = { }
C -> c C	(8)	FIRST(B) = { }
C -> ε	(9)	FIRST(C) = { }
		FIRST(D) = { }
		FIRST(E) = { }

Applying rules I and II gives us the FIRST sets of terminals and epsilon.

Initialization gives us empty FIRST sets for all non-terminals.

The sets shown above are what we get at the end of the initialization phase.

Now, we are going to start iterating over all grammar rules. This will be shown starting on the next page

Calculating FIRST sets example

S -> A B C	(1)	FIRST(ε) = { ε }
A -> B C d D E	(2)	FIRST(b) = { b }
B -> b B	(3)	FIRST(c) = { c }
B -> D E	(4)	FIRST(d) = { d }
D -> ε	(5)	FIRST(e) = { e }
E -> ε	(6)	FIRST(S) = { }
E -> e	(7)	FIRST(A) = { }
C -> c C	(8)	FIRST(B) = { }
C -> ε	(9)	FIRST(C) = { }
		FIRST(D) = { }
		FIRST(E) = { }

We examine grammar rule (1)

by rule III, we add FIRST(A) – { ε } to FIRST(S)
Since FIRST(A) is empty, there is no change

since ε ∉ FIRST(A), rule IV does not apply

rule V also does not apply

We examine grammar rule (2)

by rule III, we add FIRST(B) – { ε } to FIRST(A)
Since FIRST(B) is empty, there is no change

since ε ∉ FIRST(B), rule IV does not apply

rule V also does not apply

.... continued on next page

Calculating FIRST sets example

S -> A B C	(1)	FIRST(ε) = { ε }
A -> B C d D E	(2)	FIRST(b) = { b }
B -> b B	(3)	FIRST(c) = { c }
B -> D E	(4)	FIRST(d) = { d }
D -> ε	(5)	FIRST(e) = { e }
E -> ε	(6)	FIRST(S) = { }
E -> e	(7)	FIRST(A) = { }
C -> c C	(8)	FIRST(B) = { b[1] , }
C -> ε	(9)	FIRST(C) = { }
		FIRST(D) = { }
		FIRST(E) = { }

We examine grammar rule (3)

by rule III, we add FIRST(b) – { ε } to FIRST(B)
This adds b to FIRST(B) 1

since ε ∉ FIRST(b), rule IV does not apply

rule V also does not apply

We examine grammar rule (4)

by rule III, we add FIRST(D) – { ε } to FIRST(B)
Since FIRST(D) is empty, there is no change

since ε ∉ FIRST(D), rule IV does not apply

rule V also does not apply

.... continued on next page

Calculating FIRST sets example

S -> A B C	(1)	FIRST(ε) = { ε }
A -> B C d D E	(2)	FIRST(b) = { b }
B -> b B	(3)	FIRST(c) = { c }
B -> D E	(4)	FIRST(d) = { d }
D -> ε	(5)	FIRST(e) = { e }
E -> ε	(6)	FIRST(S) = { }
E -> e	(7)	FIRST(A) = { }
C -> c C	(8)	FIRST(B) = { b[1] , }
C -> ε	(9)	FIRST(C) = { }
		FIRST(D) = { ε[2] , }
		FIRST(E) = { ε[3] , }

We examine grammar rule (5)

by rule III, we add FIRST(ε) – { ε } to FIRST(D)
This adds nothing to FIRST(D)

rule IV does not apply because there is only one symbol on the RHS

rule V applies and we add ε to FIRST(D) 2

We examine grammar rule (6)

This will result in adding ε to FIRST(E) 3

.... continued on next page

Calculating FIRST sets example

S -> A B C	(1)	FIRST(ε) = { ε }
A -> B C d D E	(2)	FIRST(b) = { b }
B -> b B	(3)	FIRST(c) = { c }
B -> D E	(4)	FIRST(d) = { d }
D -> ε	(5)	FIRST(e) = { e }
E -> ε	(6)	FIRST(S) = { }
E -> e	(7)	FIRST(A) = { }
C -> c C	(8)	FIRST(B) = { b^1 , }
C -> ε	(9)	FIRST(C) = { c^5 , ε^6 }
		FIRST(D) = { ε^2 , }
		FIRST(E) = { ε^3 , e^4 , }

We examine grammar rule (7)

by rule III, we add FIRST(e) – { ε } to FIRST(E)
This adds e to FIRST(E) 4

rule IV does not apply and rule V does not apply

We examine grammar rule (8)

by rule III, we add FIRST(c) – { ε } to FIRST(E)
This adds c to FIRST(C) 5

rules IV and V do not apply

We examine grammar rule (9)

we add ε to FIRST(C) by rule V 6

.... continued on next page

Calculating FIRST sets example

S -> A B C	(1)	FIRST(ε) = { ε }
A -> B C d D E	(2)	FIRST(b) = { b }
B -> b B	(3)	FIRST(c) = { c }
B -> D E	(4)	FIRST(d) = { d }
D -> ε	(5)	FIRST(e) = { e }
E -> ε	(6)	FIRST(S) = { }
E -> e	(7)	FIRST(A) = { b^7 , }
C -> c C	(8)	FIRST(B) = { b^1 , }
C -> ε	(9)	FIRST(C) = { c^5 , $ε^6$ }
		FIRST(D) = { $ε^2$, }
		FIRST(E) = { $ε^3$, e^4 , }

At this point we have finished examining all the grammar rules. **Since some first sets have changed, we need to re-examine all grammar rules and apply rules III, IV, and V to them.**

We examine grammar rule (1)

Since FIRST(A) is still empty, there will be no change

We examine grammar rule (2)

by rule III, we add FIRST(B) – { ε } to FIRST(A)
This adds { b } to FIRST(A)

rules IV and V do not apply

We examine grammar rule (3)

This will add b to FIRST(B), but b is already there, so no change

Rules IV and V do not apply

.... continued on next page

Calculating FIRST sets example

S -> A B C	(1)	FIRST(ε) = { ε }
A -> B C d D E	(2)	FIRST(b) = { b }
B -> b B	(3)	FIRST(c) = { c }
B -> D E	(4)	FIRST(d) = { d }
D -> ε	(5)	FIRST(e) = { e }
E -> ε	(6)	FIRST(S) = { }
E -> e	(7)	FIRST(A) = { b^7 , }
C -> c C	(8)	FIRST(B) = { b^1 , e^8, ε^9 }
C -> ε	(9)	FIRST(C) = { c^5 , ε^6 }
		FIRST(D) = { ε^2 , }
		FIRST(E) = { ε^3 , e^4 , }

At this point we have finished examining all the grammar rules. Since some first sets have changed, we need to re-examine all grammar rules and apply rules III, IV, and V to them.

We examine grammar rule (4)

This will add e to FIRST(B) by rule IV because ε ∈ FIRST(D) 8

This will add ε to FIRST(B) by rule V because ε ∈ FIRST(D) and ε ∈ FIRST(E) 9

We examine grammar rule (5)-(9)

No change!

Since some FIRST sets have changed, we need another pass.

.... continued on next page

Calculating FIRST sets example

S -> A B C	(1)	FIRST(ε) = { ε }
A -> B C d D E	(2)	FIRST(b) = { b }
B -> b B	(3)	FIRST(c) = { c }
B -> D E	(4)	FIRST(d) = { d }
D -> ε	(5)	FIRST(e) = { e }
E -> ε	(6)	FIRST(S) = { b[10] , }
E -> e	(7)	FIRST(A) = { b[7] , }
C -> c C	(8)	FIRST(B) = { b[1] , e[8], ε[9] }
C -> ε	(9)	FIRST(C) = { c[5] , ε[6] }
		FIRST(D) = { ε[2] , }
		FIRST(E) = { ε[3] , e[4] , }

Since some FIRST sets changed in the previous pass, we need to do another pass.

We examine grammar rule (1)

We add b to FIRST(S) by rule III 10

Rules IV and V do not apply

.... continued on next page

Calculating FIRST sets example

S -> A B C	(1)	FIRST(ε) = { ε }
A -> B C d D E	(2)	FIRST(b) = { b }
B -> b B	(3)	FIRST(c) = { c }
B -> D E	(4)	FIRST(d) = { d }
D -> ε	(5)	FIRST(e) = { e }
E -> ε	(6)	FIRST(S) = { b^{10} , }
E -> e	(7)	FIRST(A) = { b^{7} , e^{11}, c^{12} , d^{13} }
C -> c C	(8)	FIRST(B) = { b^{1} , e^{8}, ε^{9} }
C -> ε	(9)	FIRST(C) = { c^{5} , ε^{6} }
		FIRST(D) = { ε^{2} , }
		FIRST(E) = { ε^{3} , e^{4} , }

We examine grammar rule (2)

We add FIRST(B) – {ε} to FIRST(A) by rule III, so
e gets added to FIRST(A) 11

Since ε ∈ FIRST(B), we add FIRST(C) – { ε } to FIRST(A)
by rule IV. This adds c to FIRST(A) 12

Since ε ∈ FIRST(B) and ε ∈ FIRST(C) ,
we add FIRST(d) – { ε } to FIRST(A) by rule IV.
This adds d to FIRST(A) 13

Since ε ∉ FIRST(d) we cannot continue applying rule IV

We examine rules (3) – (9)

no change!

Since some FIRST sets changed, we need another pass

.... continued on next page

Calculating FIRST sets example

S -> A B C	(1)	FIRST(ε) = { ε }
A -> B C d D E	(2)	FIRST(b) = { b }
B -> b B	(3)	FIRST(c) = { c }
B -> D E	(4)	FIRST(d) = { d }
D -> ε	(5)	FIRST(e) = { e }
E -> ε	(6)	FIRST(S) = { b^{10} , e^{14}, c^{14} , d^{14} }
E -> e	(7)	FIRST(A) = { b^{7} , e^{11}, c^{12} , d^{13} }
C -> c C	(8)	FIRST(B) = { b^{1} , e^{8}, ε^{9} }
C -> ε	(9)	FIRST(C) = { c^{5} , ε^{6} }
		FIRST(D) = { ε^{2} , }
		FIRST(E) = { ε^{3} , e^{4} , }

We examine grammar rule (1)

We add FIRST(A) – { ε } to FIRST(S) by rule III
This will add e , c and d to FIRST(S) (b is already there)

Rules IV and V do not apply

We examine grammar rules (2)-(9)

no change!

Since some FIRST sets changed, we need to do one more pass.

In the last pass, nothing changes and we are done.

Calculating FOLLOW sets example

S -> A B C	(1)	FIRST(ε) = { ε }	FOLLOW(S) = { $[1] , }
A -> B C d D E	(2)	FIRST(b) = { b }	FOLLOW(A) = { b[2], e[2], c[3] , }
B -> b B	(3)	FIRST(c) = { c }	FOLLOW(B) = { c[4] }
B -> D E	(4)	FIRST(d) = { d }	FOLLOW(C) = { }
D -> ε	(5)	FIRST(e) = { e }	FOLLOW(D) = { }
E -> ε	(6)	FIRST(S) = { b , e, c , d }	FOLLOW(E) = { }
E -> e	(7)	FIRST(A) = { b, e, c , d }	
C -> c C	(8)	FIRST(B) = { b , e , ε }	
C -> ε	(9)	FIRST(C) = { c , ε }	
		FIRST(D) = { ε }	
		FIRST(E) = { ε , e }	

We look at the same grammar. We start with the calculated FIRST sets as shown above.

We initialize all FOLLOW sets to empty

We add $ to FOLLOW(S) by rule I 1

Next we do one pass by applying rules IV and V to all grammar rules.

Then we repeatedly apply rules II and III to all grammar rules until there is no change

Applying rules IV and V

We start examining grammar rules

We examine rule (1) RHS

We start with A

we add FIRST(B) – { ε } to FOLLOW(A) by rule IV 2
this adds b and e to FOLLOW(A)

Since ε ∈ FIRST(B), we add FIRST(C) – {ε} to FOLLOW(A)
by rule V. This adds c to FOLLOW(A) 3

Next we look at B

we add FIRST(C) – { ε } to FOLLOW(B) by rule IV 4
this adds c to FOLLOW(B)

rule V does not apply for B

We are done with grammar rule (1)

Calculating FOLLOW sets example

S -> A B C	(1)	FIRST(ε) = { ε }	FOLLOW(S) = { $[1] , }
A -> B C d D E	(2)	FIRST(b) = { b }	FOLLOW(A) = { b[2], e[2], c[3] , }
B -> b B	(3)	FIRST(c) = { c }	FOLLOW(B) = { c[4] , d[5] }
B -> D E	(4)	FIRST(d) = { d }	FOLLOW(C) = { d[6] }
D -> ε	(5)	FIRST(e) = { e }	FOLLOW(D) = { e[7] }
E -> ε	(6)	FIRST(S) = { b , e, c , **d** }	FOLLOW(E) = { }
E -> e	(7)	FIRST(A) = { b, e, c , **d** }	
C -> c C	(8)	FIRST(B) = { b , e, ε }	
C -> ε	(9)	FIRST(C) = { c , ε }	
		FIRST(D) = { ε }	
		FIRST(E) = { ε , e }	

Applying rules IV and V

We continue examining grammar rules

We examine rule (2) RHS

We start with B

we add FIRST(C) – { ε } to FOLLOW(B) by rule IV
this results in no change

Since ε ∈ FIRST(C), we add FIRST(d) – {ε} to FOLLOW(B)
by rule V. This adds d to FOLLOW(B) 5

Since ε ∉ FIRST(d), we cannot keep on moving forward
with rule V

We continue with C

we add FIRST(d) – { ε } to FOLLOW(C) by rule IV 6

rule V does not apply

We skip d because it is a terminal

We continue with D

We add FIRST(E) – { ε } to FOLLOW(D) by rule IV
This adds e to FOLLOW(D) 7

rule V does not apply

we are done with grammar rule (2)

Calculating FOLLOW sets example

S -> A B C	(1)	FIRST(ε) = { ε }	FOLLOW(S) = { $[1] , }
A -> B C d D E	(2)	FIRST(b) = { b }	FOLLOW(A) = { b[2], e[2], c[3] , }
B -> b B	(3)	FIRST(c) = { c }	FOLLOW(B) = { c[4] , d[5] }
B -> D E	(4)	FIRST(d) = { d }	FOLLOW(C) = { d[6] }
D -> ε	(5)	FIRST(e) = { e }	FOLLOW(D) = { e[7] }
E -> ε	(6)	FIRST(S) = { b , e, c , **d** }	FOLLOW(E) = { }
E -> e	(7)	FIRST(A) = { b, e, c , **d** }	
C -> c C	(8)	FIRST(B) = { b , e , ε }	
C -> ε	(9)	FIRST(C) = { c , ε }	
		FIRST(D) = { ε }	
		FIRST(E) = { ε , e }	

Applying rules IV and V

Next we examine rules (3) through (9)

For grammar rules (3) and (5) - (9), FOLLOW set rules IV and V do not apply

For rule (4), there is no change

At this point we are done with our pass applying rules IV and V

Next, we need to do multiple iterations of applying FOLLOW sets rules II and III

Calculating FOLLOW sets example

S -> A B C (1)
A -> B C d D E (2)
B -> b B (3)
B -> D E (4)
D -> ε (5)
E -> ε (6)
E -> e (7)
C -> c C (8)
C -> ε (9)

FIRST(ε) = { ε }
FIRST(b) = { b }
FIRST(c) = { c }
FIRST(d) = { d }
FIRST(e) = { e }
FIRST(S) = { b , e, c , **d** }
FIRST(A) = { b, e, c , **d** }
FIRST(B) = { b , e , ε }
FIRST(C) = { c , ε }
FIRST(D) = { ε }
FIRST(E) = { ε , e }

FOLLOW(S) = { $[1] , }
FOLLOW(A) = { b[2] , e[2], c[3] , $[10] }
FOLLOW(B) = { c[4] , d[5] , $[9] }
FOLLOW(C) = { d[6] , $[8] }
FOLLOW(D) = { e[7], b[12] , c[12] , $[12] }
FOLLOW(E) = {b[11], e[11], c[11] , $[11] }

Applying rules II and III

We examine grammar rule (1)

by rule II, we add FOLLOW(S) to FOLLOW(C)
This adds $ to FOLLOW(C) 8

Since Since ε ∈ FIRST(), we add FOLLOW(S) to FOLLOW(B)
by rule III. This adds $ to FOLLOW(B) 9

Since Since ε ∈ FIRST(B) and ε ∈ FIRST(C) , we add
FOLLOW(S) to FOLLOW(A) by rule III. This adds $ to FOLLOW(A) 10

we are done applying rules II and III to grammar rule (1)

Next, we examine grammar rule (2)

by rule II, we add FOLLOW(A) to FOLLOW(E)
This adds b, e , c, and $ to FOLLOW(E) 11

Since Since ε ∈ FIRST(E), we add FOLLOW(A) to FOLLOW(D)
by rule III. This adds b, e, c, and $ to FOLLOW(D), but
e is already in FOLLOW(D) so it is not added again 12

We stop here because of d which is a terminal

we are done applying rules II and III to grammar rule (2)

Calculating FOLLOW sets example

S -> A B C (1)

A -> B C d D E (2)

B -> b B (3)

B -> D E (4)

D -> ε (5)

E -> ε (6)

E -> e (7)

C -> c C (8)

C -> ε (9)

FIRST(ε) = { ε }

FIRST(b) = { b }

FIRST(c) = { c }

FIRST(d) = { d }

FIRST(e) = { e }

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

FIRST(A) = { b, e, c , d }

FIRST(B) = { b , e , ε }

FIRST(C) = { c , ε }

FIRST(D) = { ε }

FIRST(E) = { ε , e }

FOLLOW(S) = { $[1] , }

FOLLOW(A) = { b[2] , e[2], c[3] , $[10] }

FOLLOW(B) = { c[4] , d[5] , $[8] }

FOLLOW(C) = { d[6] , $[7] }

FOLLOW(D) = { e[7], b[12], c[12], $[12], d[14] }

FOLLOW(E) = {b[11], e[11], c[11], $[11], d[13] }

Applying rules II and III

next we examine grammar rule (3)

no change

Next, we examine gramamr rule (4)

by rule II, we add FOLLOW(B) to FOLLOW(E)
This adds c, d, and $ to FOLLOW(E) but c and $ are already in FOLLOW(E)
so d gets added 13

Since Since ε ∈ FIRST(E), we add FOLLOW(B) to FOLLOW(D)
This adds c, d, and $ to FOLLOW(E) but c and $ are already in FOLLOW(E)
so d gets added 14

Next we examine grammar rules (5) through (9)

none of them results in any change

Since some FOLLOW sets changed in this pass, we need to do an other pass of applying FOLLOW rules II and III to all grammar rules.

In the next pass there is no change and the answer we have above the final answer.