CSE340 Fall 2019 - Homework 2

Due: Wednesday October 2 2019 by 11:59 PM on Canvas

All submissions should be typed, no exceptions.

1. Consider the grammar.

S → A B g C	(1)
A → a A f	(2)
A → (C B)	(3)
A→ D	(4)
B → b B c	(5)
B → ε	(6)
C → c C	(7)
C → ε	(8)
D → d D	(9)
D → ε	(10)

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

1.1. Calculate the FIRST for all grammar symbols as follows

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

Initialization

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

1.1.2. Do successive passes, on the grammar rules in the order they are listed and apply to each grammar rule FIRST set rules III, then FIRST set rule IV then FIRST set rule V

pass 1

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

pass 2

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

pass 3

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

pass 4 NO CHANGE

1.2. Calculate the FOLLOW sets for all non-terminals as follows

1.2.1. Do an initialization pass by applying FOLLOW sets rule I

Initialization

FOLLOW(S) = { eof }

1.2.2. Do one pass on all grammar rules, in the order they are listed, and apply to each grammar rule FOLLOW set rules IV and V.

pass 1

FOLLOW (S) = { eof }
FOLLOW (A) = { b, g, f }
FOLLOW (B) = { g, `)`, c }
FOLLOW (C) = { b, `)`}
FOLLOW (D) = { }

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

pass 2

FOLLOW (S) = { eof }
FOLLOW (A) = { b, g, f }
FOLLOW (B) = { g, `)`, c }
FOLLOW (C) = { b, `)`, eof }
FOLLOW (D) = { b, g, f }

Pass 3 NO CHANGE

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

Answer

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

1. If A → α and A → β are two grammar rules, then $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
2. If $\varepsilon \in FIRST(A)$ 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 appear on the left-hand side of more than one rules. These are non-terminals A, B, C, and D.

a) non-terminal **A**

A → a A f
A → (C B)
A→ D

FIRST(aAf) = { a }
FIRST((C B)) = { `(` }
FIRST(D) = { d , ε }

FIRST(a A f) ⋂ FIRST((C B)) = { a } ⋂ { `(` } = ∅

FIRST(a A f) ⋂ FIRST(D) = { a } ⋂ { d , ε }= ∅

FIRST((C B)) ⋂ FIRST(D) = { d , ε } ⋂ { `(` } = ∅

Condition 1 is satisfied for A.

b) non-terminal **B**

B → b B c
B → ε

FIRST(b B c) = { b }
FIRST(ε) = { ε }

FIRST(b B c) ⋂ FIRST(ε) = { b } ⋂ { ε } = ∅

Condition 1 is satisfied for B.

c) non-terminal **C**

C → c C
C → ε

FIRST(c C) = { c }
FIRST(ε) = { ε }

Hence, FIRST(c C) ⋂ FIRST(ε) = { c } ⋂ { ε } = ∅

Condition 1 is satisfied for C.

d) non-terminal **D**

D → d D
D → ε

FIRST(d D) = { c }
FIRST(ε) = { ε }

Hence, FIRST(d D) ⋂ FIRST(ε) = { d } ⋂ { ε } = ∅

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

2 . If $\varepsilon \in FIRST(\text{A})$ then $FIRST(\text{A}) \cap FOLLOW(A) = \emptyset$

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

FIRST(A) ⋂ FOLLOW(A) = {a , `(` , d , ε } ⋂ { b, g, f } = ∅
FIRST(B) ⋂ FOLLOW(B) = {b , ε } ⋂ { g, `)`, c } = ∅
FIRST(C) ⋂ FOLLOW(C) = {c , ε } ⋂ { b, `)`, eof } = ∅
FIRST(D) ⋂ FOLLOW(D) = {d , ε } ⋂ { b, g, f } = ∅

This shows that the given 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.

1.4. Write parse_S(), parse_A(), parse_B(), parse_C() and parse_D(). 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.

```
void parse_input()
{
      parse_S();
      Token t = lexer.getToken();
      if (t.type != EOF)
            syntax_error();
}

void parse_S()
{
    // S -> A B g C
    parse_A();
    parse_B();
    Token t = lexer.getToken();
    if (t.type == g_type)
        ;
    else
        syntax_error();
    parse_C();
}
```

```
void parse_A()
{   Token t = lexer.getToken();
    if (t.type == a_type)       // A -> a A f
    {
        parse_A();
        t = lexer.getToken();
        if (t.type == f_type)
            ;
        else
            syntax_error();
    } else if (t.type == LPAREN )//A ->( C B )
    {
        parse_C();
        parse_B();
        t = lexer.getToken();
        if (t.type == RPAREN)
            ;
        else
            syntax_error();
    } else if ((t.type == d_type)//A -> D ≠ ε
    {
        ungetToken(t);
        parse_D();
    } else if((t.type == b_type)|| //A-> D = ε
              (t.type == f_type) ||
              (t.type == g_type))
    {
        ungetToken(t);
        parse_D();
    } else
       syntax_error();
}
```

```
void parse_B()
{
    Token t = lexer.getToken();
    if (t.type == b_type) // B -> b B c
    {
        parse_B();
        t = lexer.getToken();
        if (t.type == c_type)
            ;
        else
            syntax_error();
    } else if ((t.type == g_type) || B -> ε
               (t.type == c_type) ||
               (t.type == RPAREN))
    {
        ungetToken(t);
    } else
        syntax_error();
}
```

```
void parse_C()
{
    Token t = lexer.getToken();
    if (t.type == c_type)  // C -> c C
    {
        parse_C();
    } else if ((t.type == b_type) || // C -> ε
               (t.type == RPAREN) ||
               (t.type == EOF))
    {
        ungetToken(t);
    } else
        syntax_error();
}

void parse_D()
{
    Token t = lexer.getToken();
    if (t.type == d_type)        // D -> d D
    {
        parse_D();
    } else if ((t.type == f_type) || // D -> ε
               (t.type == b_type) ||
               (t.type == g_type))
    {
        ungetToken(t);
    } else
        syntax_error();
}
```

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

```
parse_input()
      parse_S()
            parse_A()
                  getToken()                    // a consumed
                  parse_A()
                        getToken()              // f
                        ungetToken()            // f returned to input
                        parse_D()
                              getToken()        // f
                              ungetToken()      // f returned to input
                              ;                 // D -> ε
                        ;                       // A -> D
                  getToken()                    // f consumed
                  ;                             // A -> a A f
            parse_B()
                  getToken()                    // g
                  ungetToken()                  // g returned to input
                  ;                             // B -> ε
            getToken()                          // g consumed
            parse_C()
                  getToken()                    // c consumed
                  parse_C()
                        getToken()              // c consumed
                        parse_C()
                              getToken()        // token is d
                              syntax_error()
```

2. Consider the following operator grammar

E → E & E
E → ~ E
E → (E) | id

& is a right associative binary operator. ~ is a unary operator and has lower precedence than &

2.1. Draw the precedence table for this grammar

Most of the cells of the table are filled in the same way the table from the Spring 2019 HW3 is filled because they do not depend on the specific operators used. There are 4 cells to fill in the table that are specific to this problem. The rules are given on pages 75, 76 and 78 of the operator precedence notes.

- & ⋖· & because & is right associative
- & ⋖· ~ because ~ is a unary operator
- ~ ⋖· ~ because ~ is a unary operator
- ~ ⋖· & because & has higher precedence

We get the following table

	&	**~**	**(**	**)**	**Id**	**$**
&	⋖·	⋖·	⋖·	·⋗	⋖·	·⋗
~	⋖·	⋖·	⋖·	·⋗	⋖·	·⋗
(	⋖·	⋖·	⋖·	≐	⋖·	
)	·⋗	·⋗		·⋗		·⋗
Id	·⋗	·⋗		·⋗		·⋗
$	⋖·	⋖·	⋖·		⋖·	

2.2. Show step by step how id ~ id & id & id is parsed

Stack	**Input**	**Action**	**Justification**
$	id ~ id & id & id$	Shift	$ ⋖· id
$ id	~ id & id & id$	Reduce E → id	id ·⋗ ~
$ E	~ id & id & id$	Shift	$ ⋖· ~
$ E~	id & id & id$	Shift	~ ⋖· id
$ E~id	& id & id$	Reduce E → id	id ·⋗ &
$ E~E	& id & id$	Shift	~ ⋖· &
$ E~E&	id & id$	Shift	& ⋖· id
$ E~E&id	& id$	Reduce E → id	id ·⋗ &
$ E~E&E	& id$	Shift	& ⋖· &
$ E~E&E&	id$	Shift	& ⋖· id
$ E~E&E& id	$	Reduce E → id	id ·⋗ $
$ E~E&E& E	$	Reduce E → E&E	& ·⋗ $
$ E~E&E	$	Reduce E → E&E	& ·⋗ $
$ E~E	$	Reduce E → ~E	& ·⋗ $
$ E E	$	return	$ on stack and $ on input

This execution is according to the parser I gave you in class. The parser will not declare syntax error in this case. It simply returns. We can see that the stack has two expressions, so the input is not properly parsed as one expression and it can be declared as being invalid as one expression.

3. **(Lambda calculus binding)**. For each of the following determine for each variable x the λx. it is bound to. I have numbered the variables and the abstractions. The variables are numbered using Arabic numerals and the abstractions are numbered using roman numerals. If variables 4 and 7 are bound to abstraction I, your answer should be of the form I → 4 7 to indicate that abstraction I has variables 4, and 7 bound by it.

Example (λx. x x λx. x) x
I 1 2 II 3 4

Answer I → 1 2
II → 3

Your answers need not be colored

3.1. (λx. x x λx. x (λx. x x λx. x) x) x
I 1 2 II 3 III 4 5 IV 6 7 8

Answer I → 1 2
II → 3 7
III → 4 5
IV → 6

3.2. ((λx. x x λx. x) x (λx. x x λx. x) x) x
I 1 2 II 3 4 III 5 6 IV 7 8 9

Answer I → 1 2
II → 3
III → 5 6
IV → 7

3.3. λx. (λx. λx. x x λx. x (λx. x x) λx. x) x x
I II III 1 2 IV 3 V 4 5 VI 6 7 8

Answer I → 7 8
II →
III → 1 2
IV → 3
V → 4 5
VI → 6

4. (**Beta Reduction**) For each of the following, give the resulting expression after executing one beta reduction. If there is more than one redex in the given expression, you can choose which redex you want to reduce. If there is no reducible expression, you should say so in your answer. If the reduction requires renaming, you should do renaming first. You should show the expression after alpha renaming and then after beta reduction. You should only do renaming as needed no more than needed.

Note about answers. In the answers, I highlight the λx of the redex as well as the bound variables that will be replaced with (t') by using the red color. Also, I highlight the body of the abstraction and t' by underlining them. After the reduction, I highlight (t') by using the red color.

4.1. (λx. x (λx. x) x) (x x) (λx. x x)

Answer The expression has only one redex

$$(\ \lambda x.\ \underline{x\ (\lambda x.\ x)\ x}\ )\ \underline{(\ x\ x\ )}\ (\ \lambda x.\ x\ x\ )\ \xrightarrow{\beta}\ (\ \underline{(\ x\ x\ )\ (\lambda x.\ x)\ (\ x\ x\ )}\ )\ (\ \lambda x.\ x\ x\ )$$

4.2. x (λx. λx. x x) x

Answer The expression has no redex

4.3. (λx. (λx. (λx. x) y) z) w

Answer The expression has three redexes. I show how each of them is reduced separately

1. $(\ \lambda x.\ \underline{(\ \lambda x.\ (\ \lambda x.\ x\ )\ y\ )\ z\ )\ w}\ )\ \xrightarrow{\beta}\ (\ \lambda x.\ (\ \lambda x.\ x\ )\ y\ )\ z\ )$

   Note that no x is bound to λx. so, when we do the reduction, the body of the abstraction is unchanged, t' goes away and λx goes away.

2. $(\ \lambda x.\ (\ \lambda x.\ (\ \lambda x.\ \underline{x}\ )\ \underline{y}\ )\ z\ )\ w)\ \xrightarrow{\beta}\ (\ \lambda x.\ (\ \lambda x.\ \underline{y}\ )\ z\ )\ w)$

3. $(\ \lambda x.\ (\ \lambda x.\ \underline{(\ \lambda x.\ x\ )\ y}\ )\ \underline{z}\ )\ w)\ \xrightarrow{\beta}\ (\ \lambda x.\ (\ \underline{(\ \lambda x.\ x\ )\ y}\ )\ )\ w)$

   Note that no x is bound to λx so, when we do the reduction, the body of the abstraction is unchanged, t' goes away and λx goes away.

4.4. λz. (x. (y. z y)) y

Answer The expression has only one redex

$$\lambda z.\ (\ \lambda x.\ \underline{(\ \lambda y.\ z\ y\ )}\ )\ \underline{y}\ \xrightarrow{\beta}\ \lambda z.\ (\ \underline{(\ \lambda y.\ z\ y\ )}\ )$$

In the body of the abstraction of the redex, no x is bound to λx so, when we do the reduction, the body of the abstraction is unchanged, t' (y) goes away and λx goes away. Also, there is no need for renaming because no substitution is done.

4.5. (λx. (λy. (λz. x y) x)) y z

Answer The expression has two redexes. I show how each of them is reduced separately

1. $(\ \lambda x.\ \underline{(\ \lambda y.\ (\ \lambda z.\ x\ y\ )\ x\ )}\ )\ \underline{y}\ z\ \xrightarrow{\alpha}\ (\ \lambda x.\ \underline{(\ \lambda w.\ (\ \lambda z.\ x\ w\ )\ x\ )}\ )\ \underline{y}\ z$

   $$\xrightarrow{\beta}\ (\ \underline{(\ \lambda w.\ (\ \lambda z.\ y\ w\ )\ y\ )}\ )\ z$$

   In this example, we have to do renaming before the beta reduction because the y of t' is free and will become bound to λy if we do beta reduction without renaming

2. $(\ \lambda x.\ (\ \lambda y.\ (\ \lambda z.\ \underline{x\ y}\ )\ \underline{x}\ )\ )\ y\ z\ \xrightarrow{\beta}\ (\ \lambda x.\ (\ \lambda y.\ (\ \underline{x\ y}\ )\ )\ )\ y\ z$

4.6. (λx. (λy. (λz. x y) x) (y z))

Answer The expression has two redexes. The first one is: (λx. **(** λy. (λz. x y) x **)** (y z)) . We should do renaming first because the z in (y z) is free and will become bound to λz if we do not do renaming first.

$$(\lambda x.\ (\lambda y.\ (\lambda z.\ x\ y)\ x)\ (y\ z)) \xrightarrow{\alpha} (\lambda x.\ (\lambda y.\ (\lambda w.\ x\ y)\ x)\ (y\ z))$$

$$\xrightarrow{\beta} (\lambda x.\ (\ (\lambda w.\ x\ (y\ z)\ )\ x\ )\ )$$

The second redex is highlighted here: (λx. (λy. (λz. x y) x) (y z)), The beta reduction is straight forward and requires no renaming:

$$(\lambda x.\ (\lambda y.\ (\lambda z.\ x\ y)\ x)\ (y\ z)) \xrightarrow{\beta} (\lambda x.\ (\lambda y.\ (x\ y)\ (y\ z))$$