Caleb Klenda

Homework 3

3/23/2025

Problem 1

1.)

- Nullable: A, E
- Terminal: a, b, c, d, e
- Non-terminal: S, A, B, E

2.)

Atg File:

```
COMPILER S

CHARACTERS

TOKENS

  a = 'a' .

  b = 'b' .

  c = 'c' .

  d = 'd' .

  e = 'e' .


COMMENTS FROM "//" TO '\n'


IGNORE '\r'+'\n'+'\t'


PRODUCTIONS


S = A b | B c .

A = a A c | E | .

B = b B | d .

E = e E | .


END S.
```

Trace.txt Sets

S
first: a b d e
follow: EOF

A
first: a e
follow: b c

B
first: b d
follow: c

E
first: e
follow: b c

1.3)
Computing Select sets based on First and Follow sets:

- Select(S → A b) = {a, e, b}
- Select(S → B c) = {b, d}
- Select(A → a A c) = {a}
- Select(A → E) = {e, b}
- Select(A → EOF) = {b}
- Select(B → b B) = {b}
- Select(B → d) = {d}
- Select(E → e E) = {e}
- Select(E → EOF) = {b}

Both Select(S → A b) = {a, e, b} and Select(S → B c) = {b, d} contain b. This means the grammar is not LL(1).

1.4)

First, we can use the predictive parsing table to assume the first production (which is done since the grammar is not LL(1)).

Step 1: Initialize the Parser

- Stack: [S, EOF]
- Input: b d c EOF

Step 2: Process S

- Top of Stack: S
- Current Input Token: b
- Table Entry: S → A b (chosen because it is listed first in the productions)

- Stack After Expansion: [A, b, EOF]

Step 3: Process A

- Top of Stack: A
- Current Input Token: b
- Table Entry: A → E
- Stack After Expansion: [E, b, EOF]

Step 4: Process E

- Top of Stack: E
- Current Input Token: b
- Table Entry: E → EOF
- Stack After Expansion: [b, EOF]

Step 5: Match b

- Top of Stack: b
- Current Input Token: b
- Action: Match and consume b.
- Stack After Match: [EOF]
- Remaining Input: d c EOF

At this point, the stack is not empty, but more input remains, meaning the string is rejected

Running the input text through ATGSample also confirms this result:

```
Input text > b d c
-- line 1 col 3: EOF expected
Input text > _
```

While the grammar should accept the string, the generated parser does not resolve the LL(1) conflict correctly, and rejects the string.

Problem 2

1.)

Left associativity of +, -:

```
Enter arithmetic expression > 10 - 5 + 3
8
```

This evaluates as (10-5) – 2 → 5 -2 = 3. One would expected 2 if it were right associated.

Left associativity of *, /:

```
Enter arithmetic expression > 20 / 4 / 2
2.5
```

```
Enter arithmetic expression > 20 * 4 / 4 / 2
10
```

One would expect 10 & 40 respectively if it were right associated.

Right associativity of ^

```
Enter arithmetic expression > 2 ^ 3 ^ 2
512
```

One would expect 64 if it were left associated.

2.)

One example demonstrates this:

```
Enter arithmetic expression > 3 + 5 * 2 ^ 3 - 4 / 2
41
```

This is evaluated due to precedence as:

3 + (5 * (2 ^ 3)) - (4 / 2)

= 3 + (5 * 8) - (4 / 2)

= 3 + 40 - 2

= 41

3.) Modified .atg:

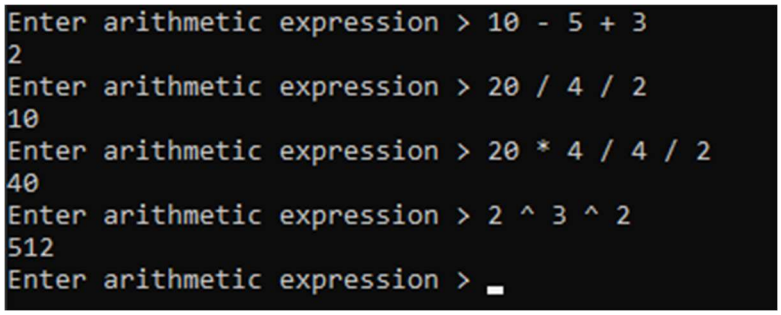
```
using Calculator;

COMPILER Calc2

CHARACTERS

    UpperLetter = 'A'..'Z'.
    LowerLetter = 'a'..'z'.
    letter = UpperLetter + LowerLetter.
    digit = "0123456789" .
    cr     = '\r' .
```

```
    lf     = '\n' .
    tab    = '\t' .

TOKENS

  ident = letter { letter | digit }.
  number = ( '.' digit {digit} [('e'|'E')  ['+'|'-'] digit {digit}])
         | ( digit { digit }'.' {digit} [('E'|'e')['+'|'-'] digit {digit}])
         | (digit {digit} ('e'|'E')  ['+'|'-'] digit {digit})
                  | digit {digit}
                  .
  div = '÷'.

IGNORE cr + tab + lf

PRODUCTIONS

Calc2                                          (. double r = 0; .)
=
Expr<out r>                          (. Console.WriteLine (r); .)
.

Expr<out double r>
=
  AddExpr<out r>
  .

AddExpr<out double r>          (. double r1; .)
=
MulExpr<out r>
{
```

```
    '+' AddExpr<out r1>             (. r += r1; .)
  |
    '-' AddExpr<out r1>             (. r -= r1; .)
}
.

MulExpr<out double r>               (. double r1; .)
=
ExpExpr<out r>
{
    '*' MulExpr<out r1>             (. r *= r1; .)
  |
    ('/' | div) MulExpr<out r1>               (. r /= r1; .)
}
.

ExpExpr<out double r>
                                                    (. double r1 = 0; .)
=
Factor<out r>
[
   '^' ExpExpr<out r1>       (. r = Math.Pow (r, r1); .)
]
.

Factor<out double r>                (.          r = 0; double sign = 1; .)
=
[
   "+"
 |
   "-"                                          (. sign = -1; .)
```

```
]
(
   number                                          (. r = sign * Convert.ToDouble (t.val); .)

  |
   '('  Expr<out r> ')'     (. r = sign * r; .)
)
.

END Calc2.
```

The following examples demonstrate the right-associativity of the new grammar.



Using “10 – 5 + 3” as an example, the key difference is that instead of immediately applying + or - on the left, it calls AddExpr recursively, ensuring that the right-most operation is evaluated first. This is the same for the multiplication and division operators. First, MulExpr<out r> parses 10 → r = 10 Then, the parser sees -, so it enters the ('+' | '-') AddExpr part and calls AddExpr<out r1>. MulExpr<out r1> parses 5 → r1 = 5. The parser sees another -, so it calls AddExpr<out r2> which sees no more symbols and returns 2. Thus, evaluating back up the it becomes 10 – (5 + 3) = 2.

Problem 3

1.)

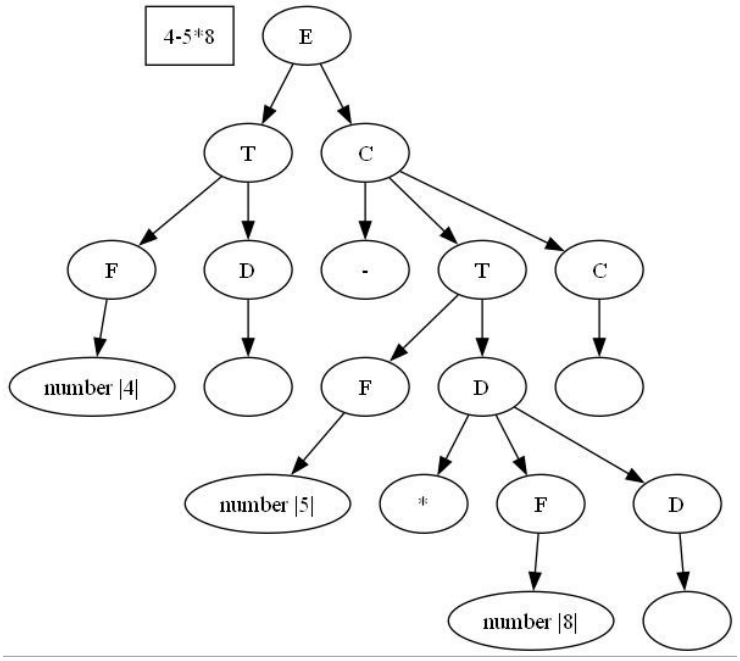
Both methods implement the same detection logic; the difference lies within the location of the recursion. In the top-most, it lies after the pruning of the tree and for the bottom-most it lies before.

The first rule checks nodes that are *internal* nodes (nodes that do not have operators) and have only one child node. These nodes are considered redundant, as they do not contribute any logical operation in the tree. The child node becomes the new child of the current node, essentially *flattening* the structure. The parent-child relationships are adjusted accordingly.

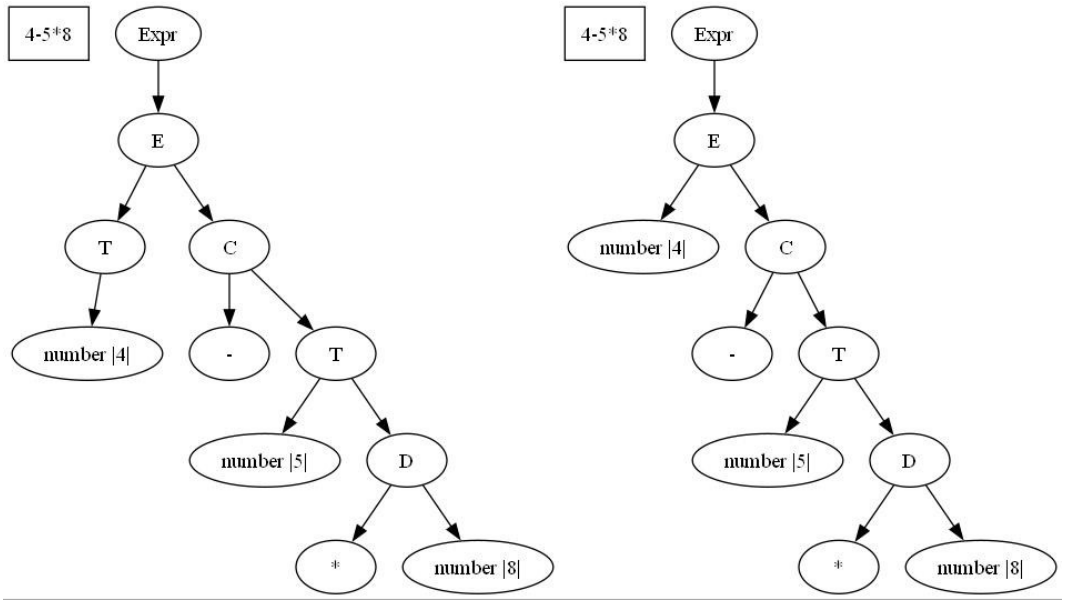
The second rule prunes leaf nodes from the tree that are non-terminal or empty. These are considered redundant since they do not contribute any terminals to the overall expression.

2.)

Unsimplified tree:



Top-down (left) and Bottom-Up (right):



The bottom-up method produces a tree with one less node. The superfluous node T in the left graph is a result of the top-down method. When the top-down method check the left-most T node, it see that its left child is F and right child is D. Since the child count it 2, it does not consider it an internal node (first elimination criteria) nor does it consider it a non-terminal leaf node (second elimination criteria). After this, the node T is considered processed, so it live on into the finally simplified tree though it carries no semantically important information.

3.)

S.atg:

```
// S1 = a A | a B .
// A = { a }.
// B = { b }.
// This should only accept "aa" and "ab"
// However, grammar is not LL(1), so we make a slight modification:
// S1 = a S2 .
// S2  = A | B .
// A  = { a } .
// B  = { b } .
```

```
using ASTClass;

COMPILER S

public AST root;

CHARACTERS

        UpperLetter = 'A'..'Z'.
        LowerLetter = 'a'..'z'.
        letter = UpperLetter + LowerLetter.
        digit = "0123456789" .

        cr      = '\r' .
        lf      = '\n' .
        tab     = '\t' .

        notQuote    = ANY - '"' - "\r\n".

TOKENS

  a = 'a' .
  b = 'b' .

COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO lf

IGNORE cr + lf + tab

//=====================================================================
```

```
PRODUCTIONS
S
                                        (. root = new AST(null,"S");
                                          TerminalAST terminal = null;                    .)
=
 S1<root>
 .

S1<AST parent>
                                        (. AST nt = new AST(parent, "S1");
                                          TerminalAST terminal = null;                    .)
=
   a (. terminal = new TerminalAST(nt, "a", t.val);  .)
        S2<nt>
        .

S2<AST parent>
                                        (. AST nt = new AST(parent, "S2");
                                          TerminalAST terminal = null;                    .)
=
        A<nt>
        |
        B<nt>
        .

A<AST parent>           (. AST nt = new AST(parent, "A");
                                          TerminalAST terminal = null;                    .)
=
   a (. terminal = new TerminalAST(nt, "a", t.val);  .)
        .

B<AST parent>           (. AST nt = new AST(parent, "B");
                                          TerminalAST terminal = null;                    .)
```

=

b (. terminal = new TerminalAST(nt, "b", t.val); .)

.

END S.

Results:

This fairly simple grammar only produces two valid strings: “aa” and “ab”. With the modified implementation, we can see the AST’s generated by parsing those strings: