Theory of Programming Languages

Example Problems

Instructor: Abdul Hameed

Write a grammar for all unsigned numbers

- Unsigned numbers are strings such as 5280, 39.37, 86E9, 6.336E4, 1.894E-3, 2.3478E+11.
- The E is essentially scientific notation, stands for exponent
- You are not allowed to have a naked decimal point.
- For example, the number 5. or .5 is not allowed.
- It must be 0.5 or 5.0 .
- The fractional part, the exponential part, and the sign on the exponential part are optional.

Grammar for all unsigned numbers

```
<number>          -->  <digit-list> <opt-fraction> <opt-exp>
<digit-list>      -->  <digit> | <digit> <digit-list>
<digit>           -->  0 | 1 | 2 | ... | 9
<opt-fraction>    -->  . <digit-list> | empty
<opt-exp>         -->  E <opt-sign> <digit-list> | empty
<opt-sign>        -->  + | - | empty
```

C switch statement

C switch statement

```
e.g.,  switch (ch) {
          case 'a': ...
          case 'b': ...
          default: ...     // optional
       }
```

Example: C switch statement

```
<switch>       --> switch ( <expr> ) { <case-list>
<opt-default> <case-list> }
<opt-default> --> empty | default: <statement>
<case-list>    --> empty | <one-case> <case-list>
<one-case>     --> case <const-value>: <statement>
<statement>    --> <block> | <assignment> | <if> |
  ...
<const-value> --> <int-const-value> | <char-const-
 value> | ...
<expr>         --> ...
```

Example: EBNF of a C switch statement

<switch_stmt> → **switch** (<expr>) {**case** <literal> : <stmt_list>
{**case** <literal> : <stmt_list> } [**default** : <stmt_list>] }

C- float literals (constants)

```
<float-literal> -->   <real> <suffix>
          |  <real> <exponent> <suffix>
          |  <integer> <exponent> <suffix>
<exponent> --> 'e' + <integer>    (note: not empty, the letter 'e' )
          |  'e' - <integer>
          |  'e' <integer>
          |  E + <integer>
          |  E - <integer>
          |  E <integer>
<integer> --> <digit> | <digit> <integer>
<real> --> .<integer>
          |  <integer> .
          |  <integer> . <integer>
<suffix> -->      f | F | empty
```

BNF of following example to give + precedence over * and force + to be right associative

Original

<assign> → <id> = <expr>

<id> → A|B|C

<expr> → <expr> + <term>
| <term>

<term> → <term> * <factor>
| <factor>

<factor> → (<expr>)
| <id>

A=A*(B+(C*A))

BNF of following example to give + precedence over * and force + to be right associative

Modified

```
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> * <term>
                | <term>
<term> → <factor> + <term>
                | <factor>
<factor> → ( <expr> )
                | <id>
```

A=A*(B+(C*A))

Left most derivation for A=A*(B+(C*A)) using following grammar

<assign>-> <id>=<expr>

<id>-> A|B|C

<expr>-> <id>+<expr>

| <id>*<expr>

| (<expr>)

| <id>

A=A*(B+(C*A))

<assign> => <id> = <expr>

=> A = <expr>

=> A = <id> * <expr>

=> A = A * <expr>

=> A = A * (<expr>)

=> A = A * (<id> + <expr>)

=> A = A * (B + <expr>)

=> A = A * (B + (<expr>))

=> A = A * (B + (<id> * <expr>))

=> A = A * (B + (C * <expr>))

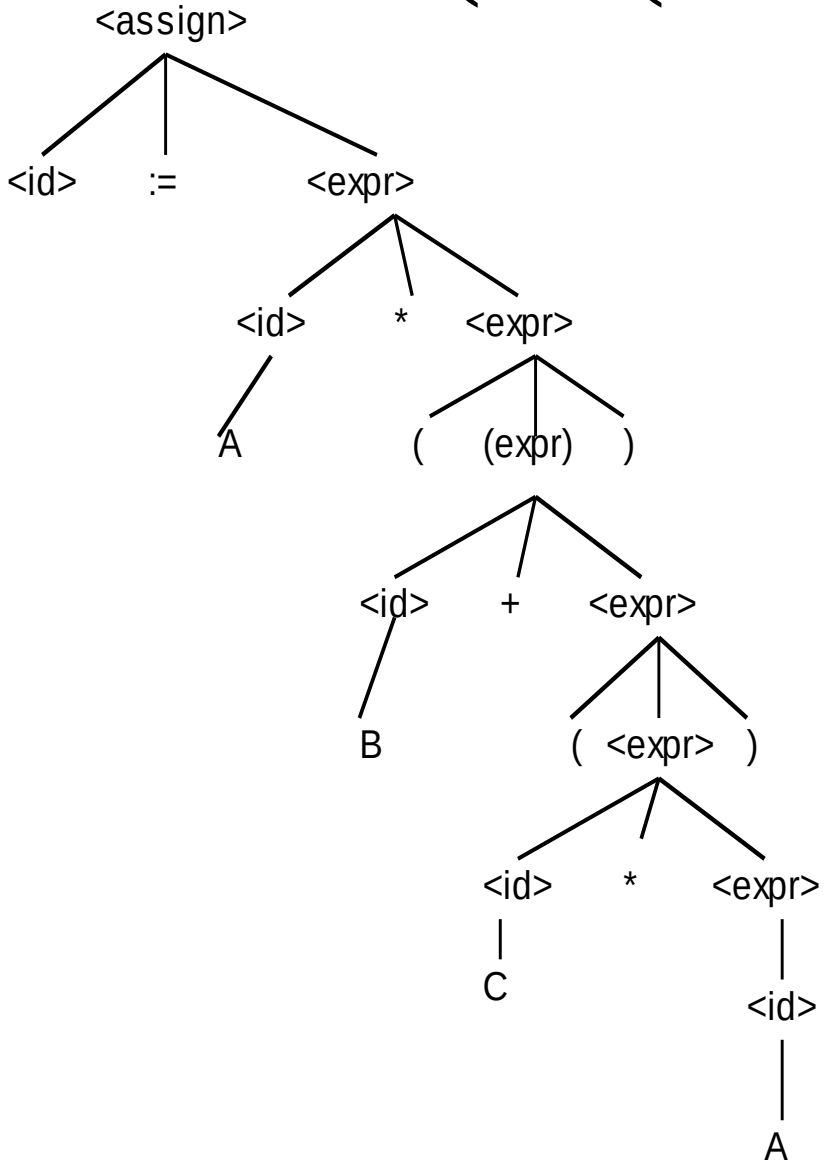
=> A = A * (B + (C * <id>))

=> A = A * (B + (C * A))

Parse tree for A=A*(B+(C*A)) using following grammar

```
<assign>-> <id>=<expr>
<id>-> A|B|C
<expr>-> <id>+<expr>
        | <id>*<expr>
        | (<expr>)
        | <id>
```

Parse tree for A=A*(B+(C*A))



Left most derivation for A=(A+B)*C using following grammar

<assign> → <id>=<expr>

<id> → A|B|C

<expr> → <expr> + <term>

| <term>

<term> → <term> * <factor>

| <factor>

<factor> → (<expr>)

| <id>

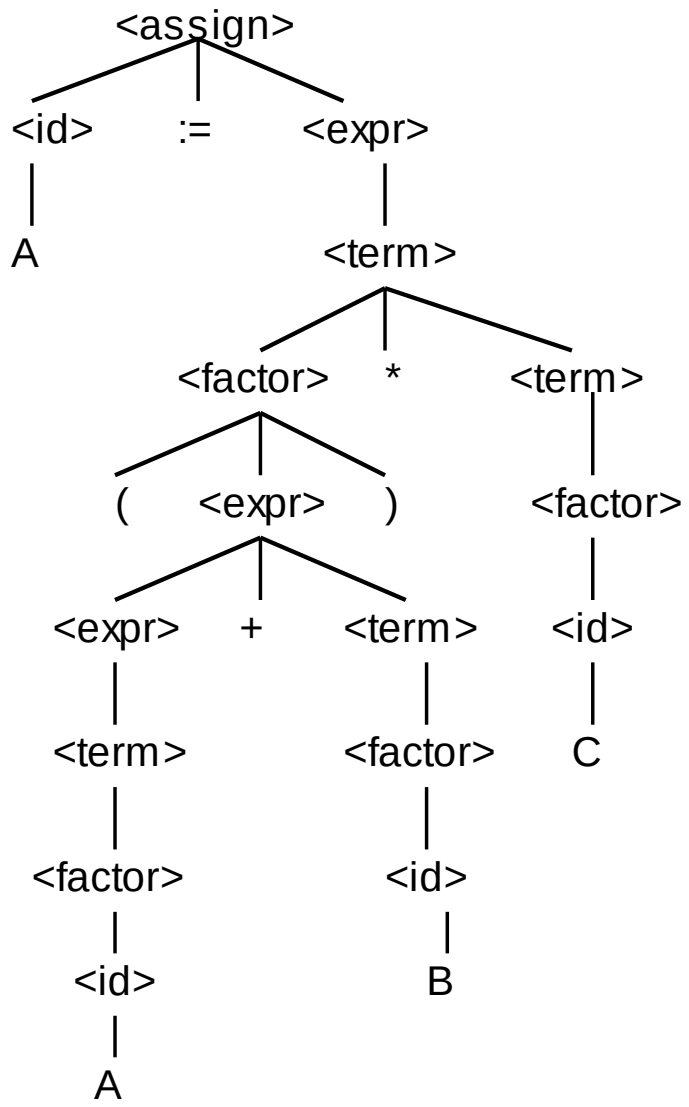
Left most derivation for A=(A+B)*C

```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <term>
         => A = <factor> * <term>
         => A = ( <expr> ) * <term>
         => A = ( <expr> + <term> ) * <term>
         => A = ( <term> + <term> ) * <term>
         => A = ( <factor> + <term> ) * <term>
```

Left most derivation for A=(A+B)*C

```
=> A = ( <id> + <term> ) * <term>
       => A = ( A + <term> ) * <term>
       => A = ( A + <factor> ) * <term>
       => A = ( A + <id> ) * <term>
       => A = ( A + B ) * <term>
       => A = ( A + B ) * <factor>
       => A = ( A + B ) * <id>
       => A = ( A + B ) * C
```

Parse tree for A=(A+B)*C



Modify the grammar to add a unary minus operator that has higher precedence than + or *

Original

```
<assign> → <id>=<expr>
<id> → A|B|C
<expr> → <expr> + <term>
         | <term>
<term> → <term> * <factor>
         | <factor>
<factor> → (<expr>)
         | <id>
```

Grammar having a unary minus operator that has higher precedence than + or *

Modified

```
<assign> → <id>=<expr>
<id> → A|B|C
<expr> → <expr> + <term>
          | <term>
<term> → <term> * <factor>
          | <factor>
<factor> → (<expr>)
          | +<id>
          | - <id>
```