# Homework 2 and 3

Programming Language Concepts

Due February 28, 2020

Name: Harry Park

1. ( points) HW2 – Describe, in English, the language defined by the following grammar:

```
<S> =: <A> <B> <C>
<A> =: a <A> | a
<B> =: b <B> | b
<C> =: c <C> | c
```

<S> is one or more a's, followed by one or more b's, followed by one or more c's.

2. ( points) HW2 – Consider the following grammar:

```
<S> =: <A> a <B> b
<A> =: <A> b | b
<B> =: a <B> | a
```

<S> is one or more b's, followed by two or more a's, followed by b.

Which of the following sentences are in the language generated by this grammar?

a. baab

b. bbbab

c. bbaaaaa

d. bbaab

3. ( points) HW2 – Explain the four criteria for proving the correctness of a logical pretest loop construct of the form "while B do S end". And prove the correctness of the following:

```
 power = 1;
 i = 1;
 while( i <= n ){
    power = power * x;
    i = i + 1;
}
{ power = x ^ n }
```

P: precondition, Q: postcondition, I: loop invariant,

while B do S end

1) P => I

2) { I and B } S { I }

3) ( I and !B ) => Q

4) The loop terminates.

1) No preconditions

2) { I and B } S { I }

```
{ n >= 0 } && { i <= n }  // i = 1
        power = power * x   => S1
        i = i + 1           => S2
{ n >= 0 }
```

```
{ n >= 0 } && { i <= n }  // i = 1
{ n > 0 }
        power = power * x   => S1
        i = i + 1           => S2
{ n >= 0 }
```

{ n > 0 }  =>  { n >= 0 }    TRUE

3) ( I and !B ) => Q

{ n >= 0 } && { i > n }
=> { power = x ^ n }  // power = 1

{ 0 <= n < i }  // i = 1
{ 0 <= n < i }
{ 0 = n }  =>  { power = x ^ n } = { power = x ^ 0 }
= { power = 1 }  =>  { 1 = 1 }

4) The loop terminates.

```
n >= 0
while ( i < = n ) {   => B
        power = power * x   => S1
        i = i + 1           => S2
}
```

```
i = 1
n > 0
while ( i < = n ) {   => B
        power = power * x   => S1
        i = i + 1           => S2
}
```

0 < n
i <= n
i < i + 1 < … < i + x

for some x where i + x = n + 1

```
i = 1
n > 0
while ( i < = n ) {   => B
        power = power * x   => S1
        i = i + 1           => S2
}
```

Never executes

4. ( points) HW2 – Give an operational semantic definition of the following:
   a. Java do-while
   b. C++ if-then-else

a. Syntax => do { statement(s); } while (expression);

Operational semantic definition =>

```
loop:
statement(s);
if (relational_expression) goto out
goto loop

out:
...
```

b. Syntax => if ( boolean_expression ) { stmt1; } else { stmt2; }

Operational semantic definition =>

```
if (boolean_expression) goto L1
goto L2
L1: stmt1; goto out
L2: stmt2;
out:
```

5. ( points) HW2 – Write a denotational semantics mapping function for the following statements:
   a. Java for
   b. Java do-while
   c. C switch

a. Java for

$M_{cf}$ (for (expr1; expr2; expr3) L, s)
    if VARMAP (i, s) = undef for some i in expr1, expr2, expr3, or L
        then error
        else if $M_e$ (expr2, $M_e$ (expr1, s)) = 0
            then s
            else $M_{help}$ (expr2, expr3, L, s)

$M_{help}$ (expr2, expr3, L, s)
    if VARMAP (I, s) = undef for some I in expr2, expr3, or L
        then error
        else
            if $M_{sl}$ (L, s) = error

b. Java do-while

$M_r$ (repeat L until B)
    if $M_b$ (B, s) = undef
        then error
        else if $M_{sl}$ (L, s) = error
            then error
            else if $M_b$ (B, s) = true
                then $M_{sl}$ (L, s)
                else $M_r$ (repeat L until B), $M_{sl}$ (L, s))

c. C switch

$M_{sw}$((expr), s)
    if VARMAP (X, s) = undef
        then error
        else VARMAP ((var), s)
    case (exp) of
        (cond_expfl) => if Mst (L, s) = error
            then error
            else Mst (L, s)
        (default_exp) => if Mst (L, s) = error
            then error

6. ( points) HW3 – Show a trace of the recursive descent parser given in "rda.c" for the following strings:

   a + b * c
   a * ( b + c )
   ( b - c ) a

1) a + b * c

Next token is: 11, Next lexeme is a
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21, Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 11, Next lexeme is b
Enter <term>
Enter <factor>
Next token is: 23, Next lexeme is *
Exit <factor>
Next token is: 11, Next lexeme is c
Enter <factor>
Next token is: -1, Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>

2) a * ( b + c )

Next token is: 11, Next lexeme is a
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 23, Next lexeme is *
Exit <factor>
Next token is: 25, Next lexeme is (
Enter <factor>
Next token is: 11, Next lexeme is b
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21, Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 11, Next lexeme is c
Enter <term>
Enter <factor>
Next token is: 26, Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: -1, Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>

3) ( b - c ) a

Next token is: 25, Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11, Next lexeme is b
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 22, Next lexeme is -
Exit <factor>
Exit <term>
Next token is: 11, Next lexeme is c
Enter <term>
Enter <factor>
Next token is: 26, Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 11, Next lexeme is a
Exit <factor>
Exit <term>
Exit <expr>
Next token is: -1, Next lexeme is EOF
Enter <expr>
Enter <term>
Enter <factor>
Error (more is desired, but not implemented).
Exit <factor>
Exit <term>
Exit <expr>

7. ( points) HW3 – Given the following grammar and the right sentential form, draw a parse tree and show the phrases and simple phrases, as well as the handle.

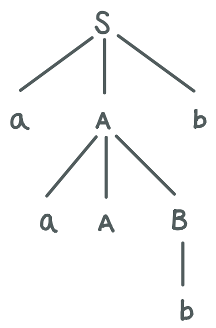
$$S \to aAb \mid bBA$$
$$A \to ab \mid aAB$$
$$B \to aB \mid b$$

a) aaAbb

b) bBab

c) aaAbBb

a) aaAbb

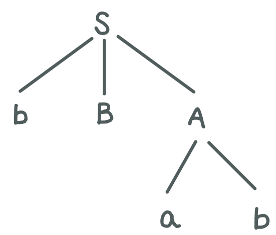



Phrases: aaAbb, aAb, b

Simple Phrase: b

Handle: b

b) bBab




Phrases: bBab, ab

Simple Phrase: ab

Handle: ab

c) aaAbBb

Not Possible

8. ( points) HW3 – Using the grammar and parse table in "LRParser.png", show a complete parse, including the parse stack contents, input string, and action for the following strings:

id * (id + id)

id * id + id

id * id / id

1) id * ( id + id )

| Stack | Input | Action |
|---|---|---|
| 0 | id*(id+id)$ | S5 |
| 0id5 | *(id+id)$ | R6 [0,F] |
| 0F3 | *(id+id)$ | R4 [0,T] |
| 0T2 | *(id+id)$ | S7 |
| 0T2*7 | (id+id)$ | S4 |
| 0T2*7(4 | id+id)$ | S5 |
| 0T2*7(4id5 | +id)$ | R6 [4,F] |
| 0T2*7(4F3 | +id)$ | R4 [4,T] |
| 0T2*7(4T2 | +id)$ | R2 [4,E] |
| 0T2*7(4E8 | +id)$ | S6 |
| 0T2*7(4E8+6 | id)$ | S5 |
| 0T2*7(4E8+6id5 | )$ | R6 [6,F] |
| 0T2*7(4E8+6F3 | )$ | R4 [6,T] |
| 0T2*7(4E8+6T9 | )$ | R1 [4,E] |
| 0T2*7(4E8 | )$ | S11 |
| 0T2*7(4E8)11 | $ | R5 [7,F] |
| 0T2*7F10 | $ | R3 [0,T] |
| 0T2 | $ | R2 [0,E] |
| 0E1 | $ | accept |

2) id * id + id

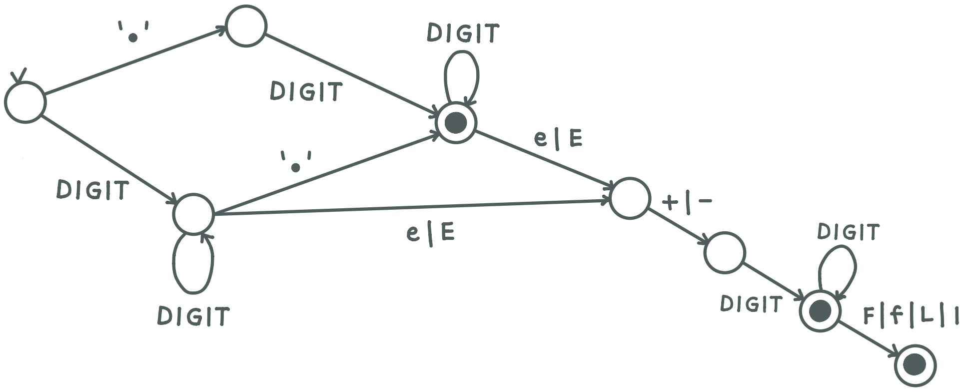
| Stack | Input | Action |
|---|---|---|
| 0 | id*id+id$ | S5 |
| 0id5 | *id+id$ | R6 [0,F] |
| 0F3 | *id+id$ | R4 [0,T] |
| 0T2 | *id+id$ | S7 |
| 0T2*7 | id+id$ | S5 |
| 0T2*7id5 | +id$ | R6 [7,F] |
| 0T2*7F10 | +id$ | R3 [0,T] |
| 0T2 | +id$ | R2 [0,E] |
| 0E1 | +id$ | S6 |
| 0E1+6 | id$ | S5 |
| 0E1+6id5 | $ | R6 [6,F] |
| 0E1+6F3 | $ | R4 [6,T] |
| 0E1+6T9 | $ | R1 [0,E] |
| 0E1 | $ | accept |

3) id * id / id

| Stack | Input | Action |
|---|---|---|
| 0 | id*id/id$ | S5 |
| 0id5 | *id/id$ | R6 [0,F] |
| 0F3 | *id/id$ | R4 [0,T] |
| 0T2 | *id/id$ | S7 |
| 0T2*7 | id/id$ | S5 |
| 0T2*7id5 | /id$ | error |

9. ( points) HW3 – Design a state diagram to recognize one floating point literals in C.

```
3.14159       /* Legal */
314159E-5L    /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */
```



10. ( points) HW3 – Design a state diagram to recognize one floating point literals in Go-Lang.

```
0.
72.40
072.40       // == 72.40
2.71828
1.e+0
6.67428e-11
1E6
.25
.12345E+5
1_5.         // == 15.0
0.15e+0_2    // == 15.0

0x1p-2       // == 0.25
0x2.p10      // == 2048.0
0x1.Fp+0     // == 1.9375
0X.8p-0      // == 0.5
0X_1FFFP-16  // == 0.1249847412109375
0x15e-2      // == 0x15e - 2 (integer subtraction)

0x.p1        // invalid: mantissa has no digits
1p-2         // invalid: p exponent requires hexadecimal mantissa
0x1.5e-2     // invalid: hexadecimal mantissa requires p exponent
1_.5         // invalid: _ must separate successive digits
1._5         // invalid: _ must separate successive digits
1.5_e1       // invalid: _ must separate successive digits
1.5e_1       // invalid: _ must separate successive digits
1.5e1_       // invalid: _ must separate successive digits
```