PLC Test1

Hyunki Lee

1. Rewriting

S-> aSb | bAA

A-> b|AaB|a|Bc

B-> a B|c

Formal definition

//nonterminals

V={<S>,<A>,<B>}

//terminals

E={a,b,c}

//Rules

R=[<S> -> a<S>b | b<A><A>

<A> -> b |<A>a<B>|a|<B>c

<B> -> a<B>|c

]

S=<S>   //Starting symbol

2. 10 smallest possible string

1) S->bAA->baA->baa

2) S->bAA->bbA->bbb

3) S->bAA->bbA->bba

4) S->bAA->baA->bab

5) S->bAA->bbA->bbBc->bbcc

6) S->bAA->baA->baBc->bacc

7) S->bAA->bBcA->bccA->bccb

8) S->bAA->bBcA->bccA->bcca

9) S->bAA->bbA->bbBc->bbaBc->bbacc

10) S->bAA->baA->baBc->baaBc->baacc

3.

a) aCbcBbcBb

S->AB->aBBB->aCbBB->aCbcBbB->aCbcBbcBb

b) ccbcbbb

S->cB->ccBb->ccCbb->ccbCAbb->ccbcAbb->ccbcbbb

c) cabbCA

S->BC->cBbC->cabC->cabbCA

4. Generate the 10 smallest possible strings

1) Forstmt->for Condition Block -> for Expression Block -> for UnaryExpr Block -> for PrimaryExpr Block -> for PrimaryExpr { StatementList } -> for PrimaryExpr { }

2) Forstmt->for Condition Block -> for Expression Block -> for UnaryExpr Block -> for unary_op UnaryExpr Block -> for + PrimaryExpr Block -> for + PrimaryExpr { StatementList } -> for + PrimaryExpr { }

3) Forstmt->for Condition Block -> for Expression Block -> for UnaryExpr Block -> for unary_op UnaryExpr Block -> for - PrimaryExpr Block -> for - PrimaryExpr { StatementList } -> for - PrimaryExpr { }

4) Forstmt->for Condition Block -> for Expression Block -> for UnaryExpr Block -> for unary_op UnaryExpr

Block -> for ! PrimaryExpr Block -> for ! PrimaryExpr { StatementList } -> for ! PrimaryExpr { }

5) Forstmt->for Condition Block -> for Expression Block -> for UnaryExpr Block -> for unary_op UnaryExpr Block -> for ^ PrimaryExpr Block -> for ^ PrimaryExpr { StatementList } -> for ^ PrimaryExpr { }

6) Forstmt->for Condition Block -> for Expression Block -> for UnaryExpr Block -> for unary_op UnaryExpr Block -> for * PrimaryExpr Block -> for * PrimaryExpr { StatementList } -> for * PrimaryExpr { }

7) Forstmt->for Condition Block -> for Expression Block -> for UnaryExpr Block -> for unary_op UnaryExpr Block -> for & PrimaryExpr Block -> for & PrimaryExpr { StatementList } -> for & PrimaryExpr { }

8) Forstmt➔for Condition Block ➔ for Expression Block ➔ for UnaryExpr Block ➔ for unary_op UnaryExpr Block ➔ for <- PrimaryExpr Block ➔ for <- PrimaryExpr { StatementList } ➔ for <- PrimaryExpr { }

9) Forstmt -> for Condition Block -> for Expression Block -> for Expression binary_op Expression Block -> for UnaryExpr || UnaryExpr Block -> for PrimaryExpr || PrimaryExpr Block -> for PrimaryExpr || PrimaryExpr { StatementList } -> for PrimaryExpr || PrimaryExpr { }

10) Forstmt-> for Condition Block -> for Expression Block -> for UnaryExpr Block -> for unary_op UnaryExpr Block -> for + unary_op UnaryExpr Block -> for + + PrimaryExpr Block -> for + + PrimaryExpr { StatementList } -> for + + PrimaryExpr { }

5. Convert the eBNF to CFG

<ForStmt> ➔ for <Condition> <Block>

<ForStmt> ➔ for <ForClause> <Block>

<ForStmt> ➔ for <RangeClause> <Block>

<Condition> ➔ <Expression>

<RangeClause>➔<ExpressionList>=range <Expression>

<RangeClause>➔<IdentifierList>:=range <Expression>

<ForClause>➔ <InitStmt>;<Condition>; <PostStmt>

<InitStmt> ➔ <SimpleStmt>

<PostStmt> ➔ <SimpleStmt>

<SimpleStmt>➔EmptyStmt | ExpressionStmt | SendStmt | IncDecStmt | Assignment | ShortVarDecl

<Expression>➔<UnaryExpr>|<Expression><binary_op><Expression>

<UnaryExpr>➔PrimaryExpr|<unary_op><UnaryExpr>

<binary_op>➔|| | && | rel_op | add_op | mul_op

<unary_op>➔+ | - | ! | ^ | * | & | <-

<Block>➔{<StatementList>}

<StatementList>➔Statement; | <StatementList>

<IdentifierList>➔identifier | ,identifier |<IdentifierList>

<ExpressionList>➔Expression| ,Expression|<ExpressionList>

Formal definition

//nonterminals

V={<ForStmt>,<Condition>,<RangeClause>,<ForClause>,<InitStmt>,<PostStmt>,<SimpleStmt>,<Expression>,<UnaryExpr>,<binary_op>,<unary_op>,<StatementList>,<IdentifierList>,<ExpressionList>}

//terminals

E={for, =, :=, range, ; ,EmptyStmt, ExpressionStmt, SendStmt, IncDecStmt, Assignment, ShortVarDecl, PrimaryExpr, ||, &&, rel_op, add_op, mul_op, +, -, !, ^, *, &, <-, Statement, , , identifier}

//Rules

R=[<ForStmt> ➔ for <Condition> <Block>

<ForStmt> ➔ for <ForClause> <Block>

<ForStmt> ➔ for <RangeClause> <Block>

<Condition> ➔ <Expression>

<RangeClause>➔<ExpressionList>=range <Expression>

<RangeClause>➔<IdentifierList>:=range <Expression>

<ForClause>➔ <InitStmt>;<Condition>; <PostStmt>

<InitStmt> ➔ <SimpleStmt>

<PostStmt> ➔ <SimpleStmt>

<SimpleStmt>➔EmptyStmt | ExpressionStmt | SendStmt | IncDecStmt | Assignment | ShortVarDecl

<Expression>➔<UnaryExpr>|<Expression><binary_op><Expression>

<UnaryExpr>➔PrimaryExpr|<unary_op><UnaryExpr>

<binary_op>➔|| | && | rel_op | add_op | mul_op

<unary_op>➔+ | - | ! | ^ | * | & | <-

<Block>➔{<StatementList>}

<StatementList>➔Statement; | <StatementList>

<IdentifierList>➔identifier | ,identifier |<IdentifierList>

<ExpressionList>➔Expression| ,Expression|<ExpressionList>

]

//Starting symbol

S=<ForStmt>

6. Find the weakest precondition

a=x/(y/3)

{B} = (x>y)

S1 = y=2x+1

S2 = y = 3x-1

{Q} = a>0

wp(IF, a>0) = (x > y Λ wp(y=2x+1, a>0)) V (x<=y Λ wp(y = 3x-1, a>0))

= (x > y Λ2x+1>a) V (x<=y Λ 3x-1>a)

=(2x+1>a) V (3x-1>a)

Therefore, a is the weakest precondition.

7. Find the weakest precondition

a=x/(y/3)

{B} = (x>y)

S1 = y=2x+1

S2 = y = 3x-1

{Q} = a>0

wp(IF, a>0) = (x > y Λ wp(y=2x+1, a>0)) V (x<=y Λ wp(y = 3x-1, a>0))

= (x > y Λ2x+1>a) V (x<=y Λ 3x-1>a)

=(2x+1>a) V (3x-1>a)

Therefore, a is the weakest precondition.

8. Prove total correctness

P->{some_num>0, i = some_num}

B->{ i != 0 }

!B->{ i == 0 }

Q->{apps = 1 + 2 + . . . + some_num}

From P and !B -> {some_num=0}

Then Q-> {apps = 0}

Thus

{some_num=i , apps=0}

```
while( i != 0 ){
        apps = apps + i;
```

```
        --i;
    }
```

From P and B ->{some_num>0} && { i !=0}

```
        {some_num>0} && { i !=0}
        apps = apps + i;
        --i;
        {some_num>0}
```

Finally checking

```
{ some_num>0, i = some_num, apps=0} -> {some_num>=0}
    while( i != 0 ){
        apps = apps + i;
        --i;
    }
    {apps = 1 + 2 + . . . + some_num}
```

9. Prove correctness

P->{x !=0, i = x/x}

B->{ x<0, i>0 }

!B->{ x>=0, i<=0 }

Q->{value = x! }

Check P and Q

{x !=0} then {i =x/x} is not zero. Thus Q->{value = x!} is valid.

Check P and B Statement Q

if loop case: assume x is negative the value will get positive number. Then i will get positive number and value will get positive number too because of { I = x/x}. x=negative, i = positive. Finally pass the while loop. Thus, Q is valid.

Check P and !B -> Q

If loop case: assume x is zero then i will be zero. From the !B {i <=0} thus, the statement { value *= i--} will be zero. Then the Q->{value = x!} is zero. As a result, the rule is satisfied.

Check loop terminate

Whatever the value of x, either positive or negative number, we can pass the if loop with {x} positive and {i} positive. {i} is positive then while loop executes before it becomes zero. Finally we can terminate while loop.

10. Draw and decorate parse tree

word = 2.0 * (5 – 10)

identifier = float * (integer - integer)

#1 Assign =: identifier = Expr

#2 Expr =: Expr + Term | Expr - Term | Term

#3 Expr1 =: Expr2 - Term [Expr1.type <==
if (Expr2.type == Term.type == integer) then integer else float]

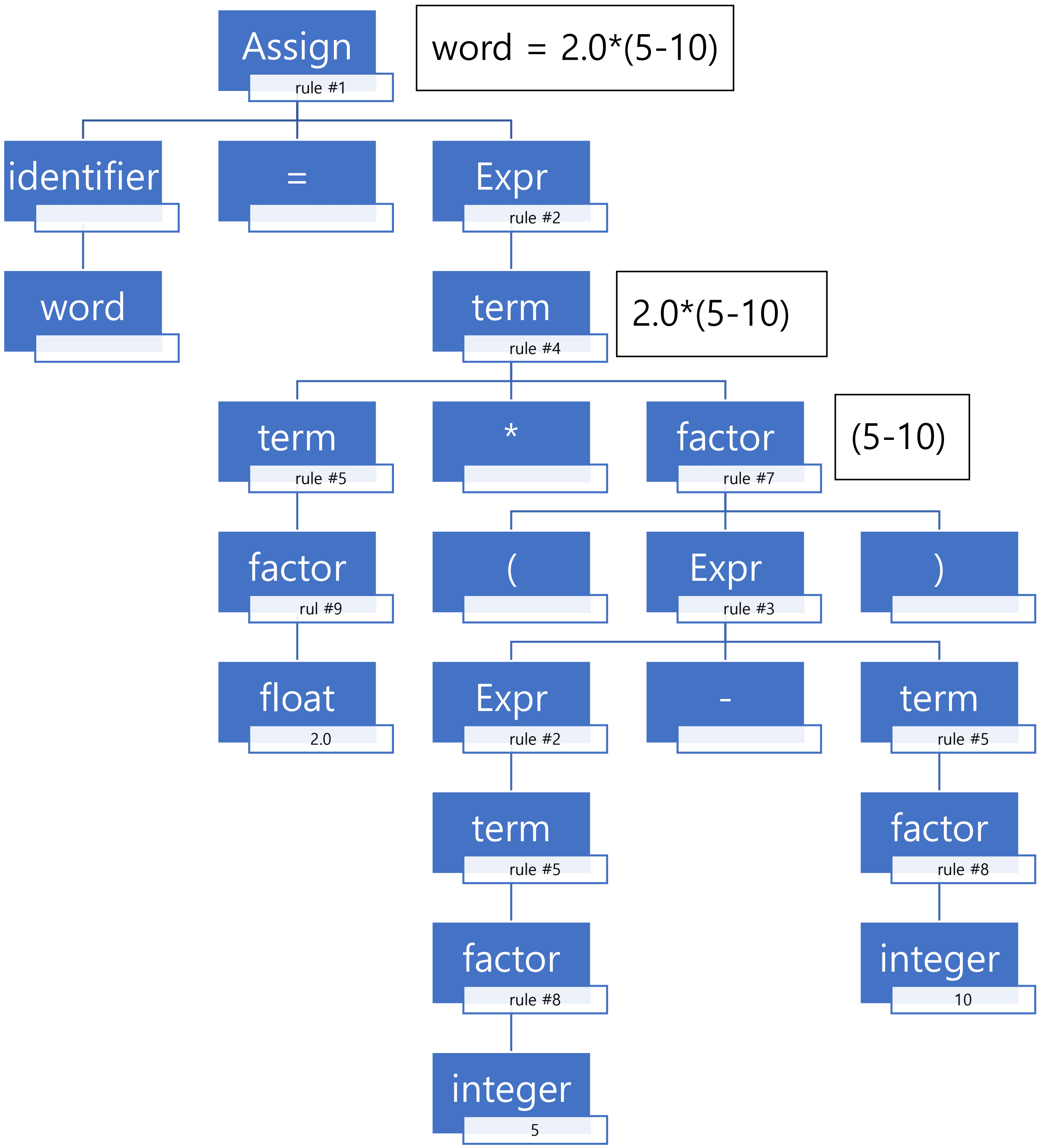
#4 Term1 =: Term2 * Factor [Term1.type <==
if (Term2.type == Factor.type == integer) then integer else float]

#5 Term =: Factor [Term.value = Factor.value]
#6 Term =: Factor [Term.type = Factor.type]

#7 Factor =: "(" Expr ")" [Factor.value = Expr.value]

#8 Factor =: integer [Factor.value = strToInt(integer.str)]
#9 Factor =: float [Factor.value =

strToFloat(float.str)]



11. Rewrit EBNF

(1) <Floating-point>= <digit-sequence> <exponent> {<suffix>}

(2) <Floating-point>=<digit-sequence> "." {<exponent>} {<suffix>}

(3) <Floating-point>= {<digit-sequence>} "." <digit-sequence> {<exponent>} {<suffix>}

(4) <Floating-point>= <hex-digit-sequence> <exponent> {<suffix>}

(5) <Floating-point>= <hex-digit-sequence> "." <exponent> {<suffix>}

(6) <Floating-point>= <hex-digit-sequence> "." <hex-digit-sequence> <exponent> {<suffix>}

(7) <digit-sequence>=a-whole-number-without-decimal-separator

(8) <digit-sequence>=a-fractional-number-with-decimal-separator

(9) <hex-digit-sequence>=whole-number-without-a-radix-separator

(10) <hex-digit-sequence>=whole-number-with-a-radix-separator

(11) < hex-digit-sequence>= a-fractional-number-with-a-radix-separator

(12) <exponent> = {<exponent-sign>}<digit-sequence>

(13) <exponent> = {<exponent-sign>}<digit-sequence>

(14) < exponent-sign> = “+” | “-“

(15) <suffix> = double | <f> | <F> | <l> | <L>

(16) <f> = float

(17) <F> = float

(18) <l>= long double

(19) <L>= long double

Rewrite CFG

(1) <Floating-point>➔ <digit-sequence> <exponent>

(2) <Floating-point>➔<digit-sequence> <exponent> <Floating-point>suffix | <suffix>

(3) <Floating-point>➔<digit-sequence> .

(4) <Floating-point> ➔ <digit-sequence> . |

<Floating-point>

(5) <Floating-point>➔ <exponent>|<Floatingpoint><exponent>|<suffix>| <Floating-point><suffix>

(6) <Floating-point>➔ . <digit-sequence>

(7) <Floating-point>➔<digit-sequence>

(8) <Floating-point> ➔ <exponent>

(9) <Floating-point> ➔ <suffix>

(10) <Floating-point>➔ <Floating-point> . <digit-sequence> <Floating-point>

(11) <Floating-point>➔ . <digit-sequence> <Floating-point><Floating-point>

(12) <Floating-point>➔<digit-sequence>|<Floating-point><digit-sequence> . <digit-sequence>

(13) <Floating-point>➔ . <digit-sequence> <exponent>|<Floating pint><exponent>

(14) <Floating-point>➔ . <digit-sequence> <suffix>|<Floating pint><suffix>

(15) <Floating-point>➔ <hex-digit-sequence> <exponent>

(16) <Floating-point>➔ <hex-digit-sequence> <exponent> <suffix>|<Floating-point><suffix>

(17) <Floating-point>➔ <hex-digit-sequence> . <exponent>

(18) <Floating-point>➔<hex-digit-sequence> . <exponent> <suffix>|<Floating-point>

(19) <Floating-point>➔ <hex-digit-sequence> . <hex-digit-sequence> <exponent>

(20) <Floating-point>➔ <hex-digit-sequence> . <hex-digit-sequence><exponent> <suffix>|<suffix><Floating-point>

(21) <digit-sequence>➔a-whole-number-without-decimal-separator

(22) <digit-sequence>➔a-fractional-number-with-decimal-separator

(23) <hex-digit-sequence>➔whole-number-without-a-radix-separator

(24) <hex-digit-sequence>➔whole-number-with-a-radix-separator

(25) < hex-digit-sequence>➔ a-fractional-number-with-

a-radix-separator

(26) <exponent> ➔ <digit-sequence>

(27) <exponent>➔<exponent-sign>|<exponent-sign><exponent> <digit-sequence>

(28) <exponent> ➔ <exponent-sign>

(29) < exponent-sign> ➔ "+" | "-"

(30) <suffix> ➔ double | <f> | <F> | <l> | <L>

(31) <f>➔ float

(32) <F>➔ float

(33) <l>➔ long double

(34) <L>➔long double

12. Esoteric programming language, write lexical analyzer. Making OREO

Code:

```
#include <stdio.h>

#include<string.h>

#define MAX 10000

int main(){
```

```
char c[MAX];

FILE *fp;

fp = fopen("Q12.txt", "r");

int i;

int size;


   if ( fp == NULL) {

   printf("Cannot open file");

   return -1;

   }


fscanf(fp, "%[^₩n]", c);

size = strlen(c);

        printf("O");

   for(i=0; i<size; i++){

      if((c[i]>='0'&&                    c[i]<='9')||(c[i]>='a'&&
c[i]<='z')||(c[i]>='A'&& c[i]<='Z')){
```

```
            printf("RE",c[i]);
        }
        else if(c[i] == ' '){
            printf("RE");
        }



        else{
            printf("&O",c[i]);
            printf("₩n");
            printf("O");
        }
    }
          printf("&O");
 fclose(fp);
 return 0;


}
```