Test One PLC Part One

Akash Dansinghani

Programming Language Concepts

Professor Umoja

Due: October 09, 2020

1.To obtain the formal definition of Context – Free – Grammar follows the four tuple (V,Σ,R,S). Defining each element we start with V which is a finite set of variables (non-terminal(s)), E or Σ is a finite set that is disjoint from V and include terminal symbols, R which contain the rules for each variable and maps to a string where the string contains V and Σ, and finally S is the start of the expression which can be found in V.

G =

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

Σ = {a,b,c}

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

<A> = b{a<B>}|a|<B>c

<B> = a<B>|c

S = <S>

2. The smallest strings that can be generated from the rules above are:

{bbb, bba, bab, baa, bcca, bacc, baca, bbcc, abcca, abaab}

Reason:

For the first three generated string, this is true since there has to be a minimum of two b's in the string and for the instance of three characters in a string, a has to follow b where b has to be in the first index of the string. For strings with four characters, b has to be in the first index of the string as well. For the string with five characters, a has to be in the first index followed by b and b has to be in between a's.

3.

S -> AB, aBBB, aCbcBbcBb

AB is in the language since it is given and is in the terminal. acBcBbcBb is fine since it follows the rule that there is a lower case b every third index

S -> cb, ccBb, ccCbb, ccbCAbb, ccbcbbb

These are generated because each string starts with a c, specifically a lower-case c and followed by a b after two c are placed consecutively in the string.

S -> BC , cabbCA, cBbbCA

A capital c has to be followed by b at the end of the string and capital "c" and "a" are at the end of the string for strings with six characters in it.

4.

Expression 1: "for": <expression> "{""}" = 4 strings

Expression 2: "for" "+" <expression> "{""}" = 5 strings

Expression 3: "for" "-" <expression> "{""}" = 5 strings

Expression 4: "for" "*" <expression> "{""}" = 5 strings

Expression 5: "for" "range" <expression> "{""}" = 5 strings

Expression 6: "for" "!" <expression> "{""}" = 5 strings

Expression 7: "for" "^" <expression> "{""}" = 5 strings

Expression 8: "for" "<-" <expression> "{""}" = 5 strings

Expression 9: "for" "&" <expression> "{""}" = 5 strings

Expression 10: "for" <"EmptyStmt"> ";" ";" "{""}" = 6 strings

5.

Converting the eBNF into CFG we start with G = (V,Σ,R,S)

V = {<ForStmt> <Range> <For> <InitStatement> <PostStmt> <SimpleStatement> <UnaryExpr> <binary_op> <unary_op> <expr><Block> <STL><EL> <IL> <Conditon>}

// STL = Statement List, EL = Expression List IL = IdentifierList SimpStatement = SimpleStatement

Σ = {indentifier, statement, ExpressionStatement, EmptyStatement, FwrdStatement, IncreaseDecreaseStatement, Assignment, ShortVarl, PrimaryExpr, "for", "=" ";", "| |", "*", "-", "+", "^", "!","<-", "{}","&"}

R = { [<ForStmt> = "for" [<Conditon>] | <For> | <Range>] <Block> <Condition> = <Expr>

<Range> = [<Expr> "=" | <IL> "="] "Range" <Expression>

<For> = [<InitStatement>] “;” [<Condition>] “;” [<PostStmt>]

<IL> = <SimpleStatement>

<PostStatement> = <SimpleStatement>

<SimpleStatement> = [EmptyStatement |ExpressionStatement|FwrdStatement|IncreaseDecreaseStatement|Assignment|ShortVarl]

<Expression> = <UnaryExpr> | <Expression><binary_op><Expression>

<UnaryExpr> = PrimaryExpr | <unary_op><UnaryExpr>

<binary_op> = “||” |“&&”| rel_op |add_op | mul_op

<unary_op> = “+” | “-“ | “*” | “^ | “!”| “&” | “<-“ | &

<Block> = “{“ <SL “}”

<SL> = { statement “;” }

<IL> = identifier { “,” identifier}

<EL> = <Expression> {“,” <Expression>}

S = <ForStmt>

6.

if (x > y)

y = 2 * x + 1

else

y = 3 * x - 1;

a = x / (y / 3);

{a must be a positive integer}

Answer: a is the weakest precondition (a = x / (y / 3)). The reason why a is the weakest precondition is because it is the last condition we need to prove and given that the if statement is passed or the else is passed, then we are left with a which makes it the weakest since it is not true in most instances.

If we are given the values x = 3 and y = 1, the if statement: if (3 > 1) is true, so we would continue to line 2. For line 2 if we substitute x = 3 and y = 1 we can see the condition is false since 2(3) = 6 +1 = 7 and since y = 1, 1 is NOT equal to 7. Thus, we execute the else statement. When substituting the values into variables x and y, we get (3)(3) = 9 – 1 = 8 and 1 is NOT equal to 1. But what we can see is after each condition is gone through, the value is incremented by 1. In the first y statement we got the value 9 and in the second y statement we got 8. Once we go to a, we can substitute x for 3 and y for 1 and we get 3 / (1 / 3) and so we can do the reciprocal and get 3(3) where we get the result 9 which is a positive integer.

So we can see in each statement, y > x after the if statement is executed and y > x after the while statement is executed. However we can see that after each statement it is incremented by 1 and we can see a > x and y if x && y > 0.

8. Prove total correctness of the following Loop:

```
{some_num > 0}

i = some_num;

while ( i != 0) {

        apps = apps + I;

        --i;

}

{apps = 1 + 2 + …. + some_num}
```

Proof:

```
{some_num > 0}

i = some_num;

apps = 0


{i= some_num, apps = 0, i > 0} ➔ { some_num >= 0}

While ( i != 0) {

        apps = apps + i

        i = i – 1

}

{apps= 1 + 2 + … + some_num}

I ➔ {some_num >= 0, some_num = i }

B ➔ { i != 0 }

!B ➔ {i == 0 }

Q ➔ { apps = 1 + 2 + … + some_num }
```

I ➔ {some_num >= 0 }

!B ➔ {i == 0 }

Some_num = 0

Q ➔ { apps = 0 }

{some_num > 0 || i!= 0}

    apps = apps + i

    i = i -1

{some_num > 0}

{I and B} ➔ {I}

{I and not B} ➔ {Q}

9. Prove the correctness of the following:

```
{x != 0}
i = x / x;
if ( x < 0 )
value = ( -x );
else
value = x;
temp = value;
value = i;
i = temp;
while( i > 0 ){
value *= i--;
}
{value = x!}
```

Proof:

```
{x != 0}
i = x / x;
{x > 0,  i= x/x, } ➔ { i >= 0}

if ( x < 0 )
value = ( -x );
        i = i – 1
        {x < 0,  value = -x } ➔ { value < 0}

I ➔ {x != 0, i = x/x }
B ➔ { i > 0 }
!B ➔ {i == 0 }
Q ➔ {value = x}

I ➔ {x < 0}
B ➔ { value < 0 }
!B ➔ {value >= 0 }
Q ➔ { value = -x}
else
value = x;
temp = value;
value = i;
i = temp;
I ➔ { (i > 0) }
!B ➔ {i <= 0 }
Q ➔ { value *= i--;}
```

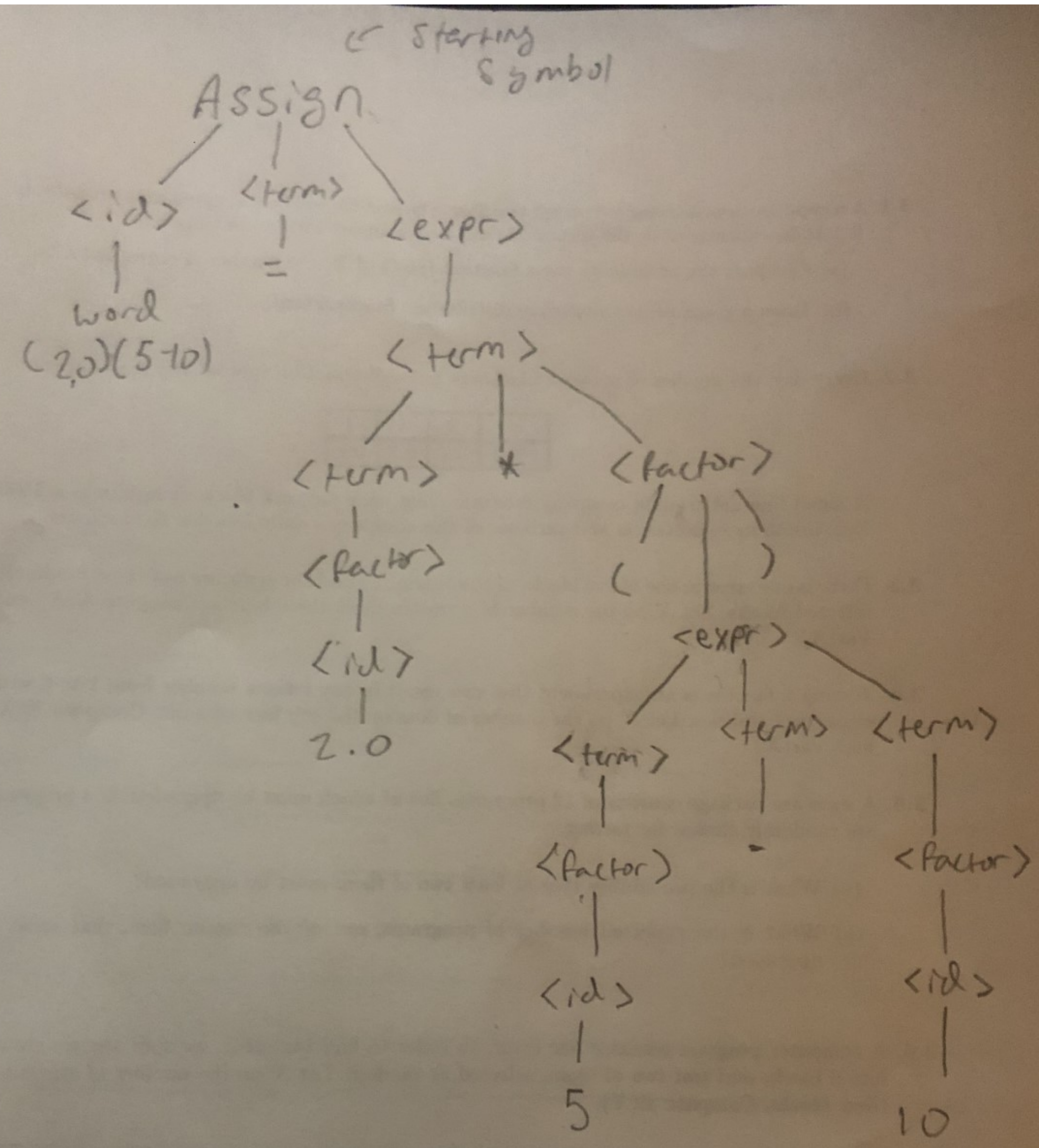
```
while( i > 0 ){
value *= i--;
}
{value = x!}
```

{I and B} ➔ {I}

{I and not B} ➔ {Q}

Explanation of Proof: if x does not equal 0, it can be divided by each other and get a whole number as well as a positive number regardless of it being a positive or negative integer. Then for the if statement it says if x < 0, then value would be = -x (However if x is already a negative number, we can just use value = x, but we are trying to make it positive so we can continue to the else loop). Now, value = x. Now we change contents of value which is x and make it equal to temp. So temp holds the contents of value. Now value holds i. Finally i holds temp which is x. So while (i > 0), value will be multiplied by I that is decremented until value equals a negative number.

10.


← Starting symbol
Assign
<id>
word
(2.0)(5-10)
<term>
=
<expr>
<term>
<term>
*
<factor>
<factor>
<id>
2.0
(
)
<expr>
<term>
<term>
<term>
-
<factor>
<factor>
<id>
<id>
5
10

11.

EBNF

Digit = digit – sequence | exponent | [suffix]

Digit = digit – sequence | [exponent] | [suffix]

Digit = [digit – sequence] | “.” digital – sequence | [exponent] | [suffix]

Hex – digit sequence = 0x | 0x hex – digit – sequence | exponent | [suffix]

Hex – digit sequence = 0x | 0x hex – digit – sequence | “.” | exponent | [suffix]

Hex – digit sequence = 0x | 0x [hex – digit – sequence] | “.” | hex – digit – sequence | exponent | [suffix]

Digit – sequence = integer | “.”? | exponent ::= 1e10, 1e – 5L

Digit – sequence = integer + “.” | [exponent] ::= “1. , 1.e -2”

Digit-sequence = fractional number | “.” | [exponent] ::= 3.14, .1f, 0.1e-1L

Hexadecimal digit-sequence = integer + radix separator? | exponent | hexadecimal floating-point literals ::= 0x1ffp10, 0X0p-1

Hexadecimal digit-sequence = integer + radix separator | “.” | exponent | hexadecimal floating-point literals ::= 0x1.p0, 0xf.p-1

Hexadecimal digit-sequence = fractional number + radix separator | “.” | exponent | hexadecimal floating-point literals ::= 0x0.123p-1, 0xa.bp10l

Exponent = e | E [exponent - sign] digit - sequence

Exponent = p | P [exponent – sign] digit – sequence

EBNF Syntax and Key

Digit – sequence ::= digit | [digit] ;

Hex – digit – sequence ::= 0x | 0x ;

Hexadecimal - digit ::= [0-9A-Fa-f]= fractional constant | [part] | [suffix] ;

integer ::= [+-]? | [0-9] +| digit sequence, | exponent part | [suffix] ;

fractional number ::= [digit sequence], ".", digit sequence | digit sequence, "." ;

exponent ::= ("e" | "E") | [sign] | digit sequence ;

exponent – sign ::= [+] | [-] | exponent ;

sign ::= "+" | "-" ; digit sequence = digit | digit sequence | digit ;

suffix ::= "f" | "l" | "F" | "L" ;

digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

double ::= 19 * digit, [[, | .] 18 * digit]["d" | "D"];

float ::= 19 * digit, [[, | .] 18 * digit, "f" | "F"];

CFG

To form the CFG of the syntax in the website, we have to use the four tuple:

G =

V = {<rule>, <digit>, <digit-sequence>, <hexadecimal-digit-sequence>, <suffix>, <exponent>, <exponent-sign>, <integer>, <fractional>, <decimal>, <hexadecimal>, <radix-separator>, <addition>, <subtraction>}

Σ = {A,B,C,D,E,e,f,F,p,P,l,L,+,-, “.”, 0x, 0X, 0,1,2,3,4,5,6,7,8}

R =

{

<digit> = <digit – sequence> | <exponent> | { <suffix> }

<digit> = {<digit – sequence>} | <decimal> | { <exponent> } | { <suffix> }

<digit> = {<digit – sequence>} | <decimal> | {<digital – sequence>} | { <exponent> } | { <suffix> }

<hexadecimal-digit-sequence> = { 0x | 0X | < hexadecimal - digit – sequence>} | <exponent> | { <suffix> }

<hexadecimal-digit-sequence> = { 0x | 0X | < hexadecimal - digit – sequence>} | <decimal> | <exponent> | { <suffix> }

<hexadecimal-digit-sequence> = { 0x | 0X | {< hexadecimal - digit – sequence>} | <decimal> | < hexadecimal - digit – sequence> | <exponent> | { <suffix> }

<digit-sequence> = {<integer> | <exponent> }

<digit-sequence> = {<integer> |<decimal> | { <exponent> }

<digit-sequence> = {<fractional> | { <exponent> }

<hexadecimal-digit-sequence> = {<integer> | <exponent> }

<hexadecimal-digit-sequence> = {<integer> | <radix-separator> | <exponent> }

<hexadecimal-digit-sequence> = {<fractional> | <radix-separator> | <exponent> }

<exponent> = { e | E | <exponent-sign> } | <digital-sequence> }

<exponent> = { p | P |<exponent-sign> } | <digital-sequence> }

<exponent-sign> = { <addition> | <subtraction> }

}

S = <digit>

CFG Syntax and Key

Digit – sequence = {digit | {<digit> }

Hex – digit – sequence ::= {0x | 0x }

Hexadecimal - digit ::= { [0-9A-Fa-f]= fractional constant | [part] | [suffix] }

integer ::= { [+-]? [0-9]+ | digit sequence | exponent part | [suffix] }

fractional number= {[digit sequence], ".", digit sequence | digit sequence | "." ;}

exponent = {("e" | "E") | [sign] | digit sequence}

exponent – sign = { [+] | [-] | exponent}

sign = { "+" | "-" ; digit sequence = digit | digit sequence, digit}

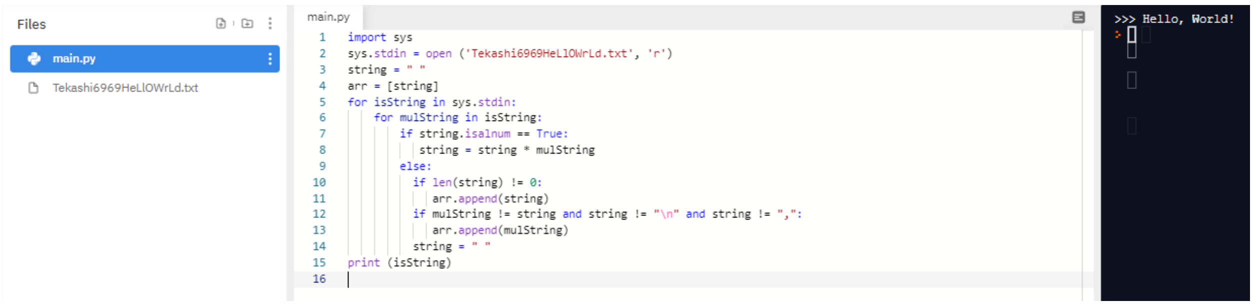
suffix = { "f" | "l" | "F" | "L" }

digit = { "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" }

double = { 19 * digit, [[, | .] 18 * digit]["d" | "D"] }

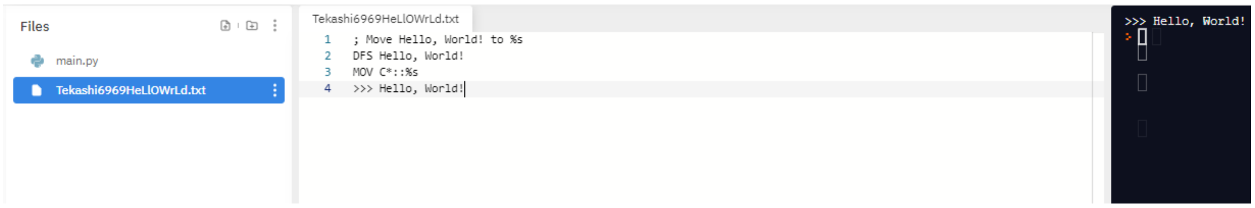
float = { 19 * digit, [[, | .] 18 * digit, "f" | "F"] }

12. Files uploaded in iCollege



```python
import sys
sys.stdin = open ('Tekashi6969HeLlOWrLd.txt', 'r')
string = " "
arr = [string]
for isString in sys.stdin:
    for mulString in isString:
        if string.isalnum == True:
            string = string * mulString
        else:
          if len(string) != 0:
              arr.append(string)
          if mulString != string and string != "\n" and string != ",":
              arr.append(mulString)
          string = " "
print (isString)
```

```
>>> Hello, World!
```



```
; Move Hello, World! to %s
DFS Hello, World!
MOV C*::%s
>>> Hello, World!
```

```
>>> Hello, World!
>
```

Citations

"6969 Assembler," *Esolang*. [Online]. Available: https://esolangs.org/wiki/6969_Assembler. [Accessed: 07-Oct-2020].

Department of Computer Science - Old Dominion University. [Online]. Available: https://www.cs.odu.edu/~toida/nerzic/390teched/cfl/cfg.html. [Accessed: 07-Oct-2020].

"EBNF: How to describe the grammar of a language," *Strumenta*, 07-Aug-2020. [Online]. Available: https://tomassetti.me/ebnf/. [Accessed: 07-Oct-2020].