Lab 2

Yuxiang Wang

Principles of Programming Languages

CSCI 3155-Fall 2017

2. Grammars: Synthetic Examples.

(a) Describe the language defined by the following grammar:

S ::= A B A

A ::= a | a A

B ::= ε | b B c | B B

Answer：we know ‘S’ is defined by ‘A', followed by ‘B’, then followed by ‘A’, and we also know ‘A’ ::= a | a A , it can be a{a, aa, aaa, aaaa…….} and B ::= ε | b B c | B B, it an be {ε, bc, bbcc, bbbccc…bcbc, bcbcbcbc, bbccbbcc……bcbbcc….},then ‘S’ combine ‘A’ and ‘B’ by ABA.

(b) Consider the following grammar:

S ::= A a B b

A ::= A b | b

B ::= a B | a

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

the sentences that are described by this grammar, demonstrate that they are by giving

Lab 2

derivations.

1. baab

2. bbbab

3. bbaaaaa

4. bbaab

(from Sebesta, Chapter 3)

Answer:

1. baab

S ::= AaBb => baBb because (A ::= A b | b);

=> baab because (B ::= a B | a);

So baab IS IN the language

2. bbbab

S ::= AaBb => AbaBb because (A ::= A b | b);

=> AbbaBb

=> bbbaBb because (A ::= A b | b);

=> bbbaab because (B ::= a B | a);

Because B ::= a B | a, so we can not get the bbbab for sure , so bbbab IS NOT IN the language

3.bbaaaaa

S ::= AaBb => AbaBb because (A ::= A b | b);

=> bbaBb because (A ::= A b | b);

=> bbaaBb because (B ::= a B | a);

=> bbaaaBb => because (B ::= a B | a);

Because bbaaaaa is end by ‘a’, but we know ‘S’ is end by ‘b’, so this language can not end with ‘a’ for sure, so bbaaaaa IS NOT IN the language generated by the above grammar.

4. bbaab

S ::= AaBb => AbaBb because (A ::= A b | b);

=> bbaBb because (A ::= A b | b);

=> bbaab (B ::= a B | a);

So baab IS IN the language.

(c) Consider the following grammar:

S ::= a S c B | A | b

A ::= c A | c

B ::= d | A

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

the sentences that are described by this grammar, demonstrate that they are by giving

parse trees.

1. abcd

2. acccbd

3. acccbcc

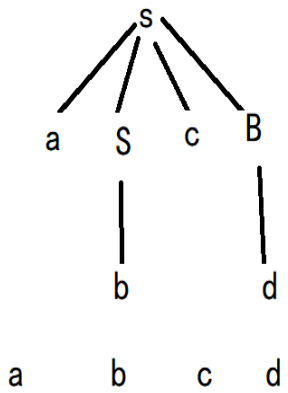
Lab 2

4. acd

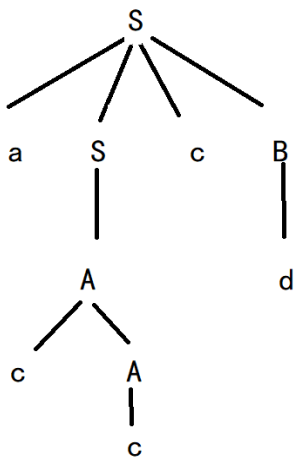
5. accc

(from Sebesta, Chapter 3)

1. abcd - Is in the language

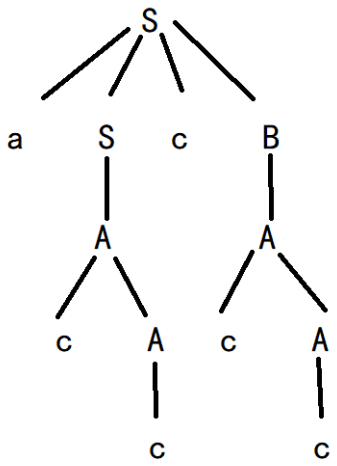


2. acccbd - Not in the langauge

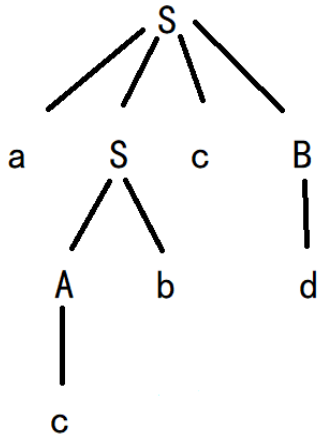


Lab 2

3. acccbcc - Not in the language

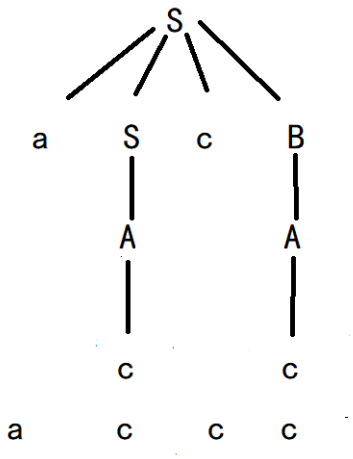


4. acd - Not in the language



Lab 2

5. accc - In the language

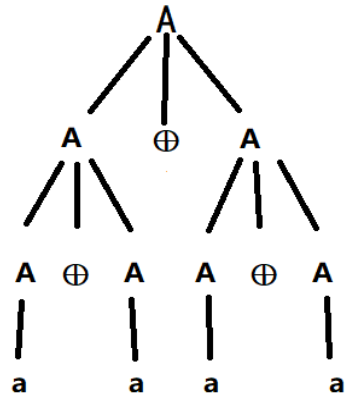


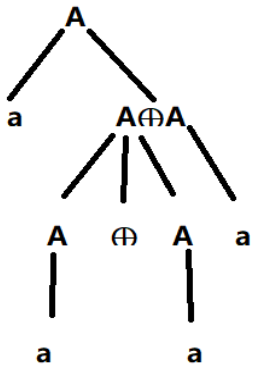
(d) Consider the following grammar:

A ::= a | b | A⊕A

Show that this grammar is ambiguous.

Answer:





Lab 2

From the graph, we can see two parse tree have the same expression, both tree can express “aaaa”, so this grammar is ambiguous.

(e) Let us ascribe a semantics to the syntactic objects A specified in the above grammar from part d. In particular, let us write

$$A \Downarrow n$$

for the judgment form that should mean A has a total n a symbols where n is the metavariable for natural numbers. Define this judgment form via a set of inference rules. You may rely upon arithmetic operators over natural numbers. Hint: There should be one inference rule for each production of the non-terminal A (called a syntax-directed judgment form).

Answer: A has n little ’a’ s and “b”s.

$$\frac{}{a \Downarrow 1}$$

$$b \Downarrow 0$$

$$\frac{A1 \Downarrow n1 \ , \ A2 \Downarrow n2}{A1 \oplus A2 \Downarrow n1 + n2}$$

$$\frac{\begin{array}{l} A1 \Downarrow n1 \\ A2 \Downarrow n2 \\ n1 + n2 = n3 \end{array}}{A1 \oplus A2 \Downarrow n3}$$

Lab 2

---

3. Grammars: Understanding a Language.

(a) Consider the following two grammars for expressions e. In both grammars, operator

and operand are the same; you do not need to know their

productions for this question.

e ::= operand | e operator operand

e ::= operand esuffix

esuffix ::= operator operand esuffix | ε

i. Intuitively describe the expressions generated by the two grammars.

ii. Do these grammars generate the same or different expressions? Explain.

Answer:

(i) The first expression is left associative because the recursive non-terminal symbol is on the left of the parse tree. grammar is produces either 'operand' or 'e operator operand' and recursively calls e, like: OPERAND OPERATOR OPERAND OPERATOR OPERAND ...we keep adding on the left.

The second expression is right associative because the recursive non-terminal symbol is on the right of the parse tree. Like OPERAND OPERATOR OPERAND OPERATOR OPERAND ... we keep adding on the right.

Lab 2

(ii): There two grammars generate the same expressions, first one use left associative and second one use right associative, but these grammars don't generate the same tree.

The first one is left recursively, give us OPERAND OPERATOR OPERAND OPERATOR OPERAND ... OPERATOR OPERAND, because the "e" is at left of expression , so we only can terminate in operand. The second one is right recursively, should be the same answer because esuffix is on the right, and second part give us OPERATOR OPERAND OPERATOR OPERAND…then put it in the first part is OPERAND OPERATOR OPERAND OPERATOR OPERAND ... OPERATOR OPERAND.

(b) Write a Scala expression to determine if '−' has higher precedence than '<<' or vice versa. Make sure that you are checking for precedence in your expression and not for left or right associativity. Use parentheses to indicate the possible abstract syntax trees, and then show the evaluation of the possible expressions. Finally, explain how you arrived at the relative precedence of '−' and '<<' based on the output that you saw in the Scala interpreter

Answer:

val expression = 5 - 1 << 2

Lab 2

```
val minus first = (5 - 1) << 2
val shift first = 5 - (1 << 2)
if (expression == minus first)
    println("- has higher precedence over <<")
if (expression == shift first)
    println("<< has higher precendence over -")
```

From the code, we know5 - 1 << 2 = 16 and (5-1)<<2=16 and5 - (1 << 2)=1, so we can see minus first = expression > shift first. Therefore, this shows us that - has higher precedence

over <<.

(c) Give a BNF grammar for floating point numbers that are made up of a fraction (e.g., 5.6 or 3.123 or -2.5) followed by an optional exponent (e.g., E10 or E-10). The exponent, if it exists, is the letter 'E' followed by an integer. For example, the following are floating point numbers: 3.5E3, 3.123E30, -2.5E2, -2.5E-2, and 3.5. The following are not examples of floating point numbers: 3.E3, E3, and 3.0E4.5. More precisely, our floating point numbers must have a decimal point, do not have leading zeros, can have any number of trailing zeros, non-zero exponents (if it exists), must have non-zero fraction to have an exponent, and cannot have a '-' in front of a zero number. The exponent cannot have leading zeros. For this exercise, let us assume that the tokens are characters in the following alphabet

Lab 2

Σ: Σ def = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,E, -,.}

Your grammar should be completely defined (i.e., it should not count on a non-terminal that it does not itself define).

Answer:

d ::= 0|1|2|3|4|5|6|7|8|9

n ::= d | dn

i :: z | -z

c ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

z ::= c | cn

f ::= i.n | i.nEip

p :: = empty | d|