**Compiler**

**Exercises for Chapter 2**

# Exercises for Section 2.2

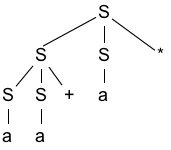
## 2.2.a

Consider the context-free grammar:

S -> S S + | S S \* | a

1. Show how the string aa+a\* can be generated by this grammar.
2. Construct a parse tree for this string.
3. What language does this grammar generate? Justify your answer.

### Answer

1. S -> S S *-> S S + S* -> a S + S *-> a a + S* -> a a + a \*
2.  3. L = {Postfix expression consisting of digits, plus and multiple signs}

## 2.2.b

What language is generated by the following grammars? In each case justify your answer.

1. S -> 0 S 1 | 0 1
2. S -> + S S | - S S | a
3. S -> S ( S ) S | ε
4. S -> a S b S | b S a S | ε
5. S -> a | S + S | S S | S \* | ( S )

### Answer

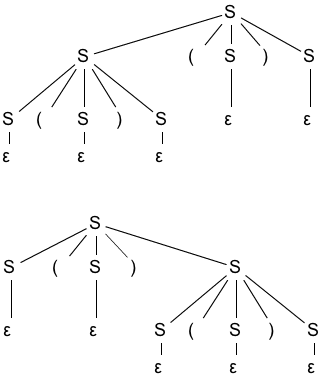
1. L = {0n1n | n>=1}
2. L = {Prefix expression consisting of plus and minus signs}
3. L = {Matched brackets of arbitrary arrangement and nesting, includes ε}
4. L = {String has the same amount of a and b, includes ε}
5. L = {Regular expressions used to describe regular languages}

## 2.2.c

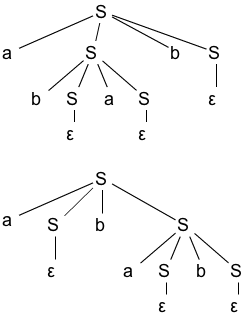
Which of the grammars in Exercise 2.2.2 are ambiguous?

### Answer

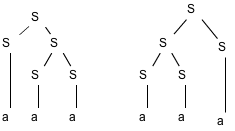
1. No
2. No
3. Yes



1. Yes



1. Yes



## 2.2.d

Construct unambiguous context-free grammars for each of the following languages. In each case show that your grammar is correct.

1. Arithmetic expressions in postfix notation.
2. Left-associative lists of identifiers separated by commas.
3. Right-associative lists of identifiers separated by commas.
4. Arithmetic expressions of integers and identifiers with the four binary operators +,

-, \*, /.

1. Add unary plus and minus to the arithmetic operators of 4.

### Answer

|  |
| --- |
| 1. E -> E E op | num      1. list -> list , id | id 2. list -> id , list | id      1. expr -> expr + term | expr - term | term term -> term \* factor | term / factor | factor   factor -> id | num | (expr)     1. expr -> expr + term | expr - term | term term -> term \* unary | term / unary | unary unary -> + factor | - factor | factor factor - > id | num | (expr) |

## 2.2.e

1. Show that all binary strings generated by the following grammar have values divisible by 3. Hint. Use induction on the number of nodes in a parse tree.

num -> 11 | 1001 | num 0 | num num

1. Does the grammar generate all binary strings with values divisible by 3?

### Answer

1. Proof

Any string derived from the grammar can be considered to be a sequence consisting of 11 and 1001, where each sequence element is possibly suffixed with a 0.

Let n be the set of positions where 11 is placed. 11 is said to be at position i if the first 1 in 11 is at position i, where i starts at 0 and grows from least significant to most significant bit.

Let m be the equivalent set for 1001.

The sum of any string produced by the grammar is:

sum

= Σn (21 + 20) *2 n + Σm (23 + 20)* 2m

= Σn 3 *2 n + Σm 9* 2m

This is clearly divisible by 3.

1. No. Consider the string "10101", which is divisible by 3, but cannot be derived from the grammar.

Readers seeking a more formal proof can read about it below:

**Proof**:

Every number divisible by 3 can be written in the form 3k. We will consider k > 0 (though it would be valid to consider k to be an arbitrary integer).

Note that every part of num(11, 1001 and 0) is divisible by 3, if the grammar could generate all the numbers divisible by 3, we can get a production for binary k from num's production:

|  |  |
| --- | --- |
| 3k = num -> 11 | 1001 | num 0 | num num k = num/3 -> 01 | 0011 | k 0 | k k k -> 01 | 0011 | k 0 | k k | |
| It is obvious that any value of k | that has more than 2 consecutive bits set to 1 can |

never be produced. This can be confirmed by the example given in the beginning:

10101 is 3\*7, hence, k = 7 = 111 in binary. Because 111 has more than 2 consecutive 1's in binary, the grammar will never produce 21.

## 2.2.6

Construct a context-free grammar for roman numerals.

**Note:** we just consider a subset of roman numerals which is less than 4k.

### Answer

[wikipedia: Roman\_numerals](http://en.wikipedia.org/wiki/Roman_numerals)

* via wikipedia, we can categorize the single roman numerals into 4 groups:

|  |  |
| --- | --- |
|  | I, II, III | I V | V, V I, V II, V III | I X |

then get the production:

|  |
| --- |
| digit -> smallDigit | I V | V smallDigit | I X smallDigit -> I | II | III | ε |

* and we can find a simple way to map roman to arabic numerals. For example:
  + XII => X, II => 10 + 2 => 12
  + CXCIX => C, XC, IX => 100 + 90 + 9 => 199
  + MDCCCLXXX => M, DCCC, LXXX => 1000 + 800 + 80 => 1880  via the upper two rules, we can derive the production:

romanNum -> thousand hundred ten digit thousand -> M | MM | MMM | ε

hundred -> smallHundred | C D | D smallHundred | C M smallHundred -> C | CC | CCC | ε ten -> smallTen | X L | L smallTen | X C smallTen -> X | XX | XXX | ε digit -> smallDigit | I V | V smallDigit | I X smallDigit -> I | II | III | ε

# Exercises for Section 2.3

## 2.3.a

Construct a syntax-directed translation scheme that translates arithmetic expressions from infix notation into prefix notation in which an operator appears before its operands; e.g. , -xy is the prefix notation for x - y. Give annotated parse trees for the inputs 9-5+2 and 9-5\*2.

### Answer

productions:

|  |
| --- |
| expr -> expr + term | expr - term | term term -> term \* factor | term / factor | factor factor -> digit | (expr) |

translation schemes:

|  |
| --- |
| expr -> {print("+")} expr + term | {print("-")} expr - term  | term  term -> {print("\*")} term \* factor | {print("/")} term / factor  | factor  factor -> digit {print(digit)}  | (expr) |

## 2.3.b

Construct a syntax-directed translation scheme that translates arithmetic expressions from postfix notation into infix notation. Give annotated parse trees for the inputs 952 *and 952*-.

### Answer

productions:

|  |
| --- |
| expr -> expr expr + | expr expr -  | expr expr \*  | expr expr /  | digit |

translation schemes:

|  |
| --- |
| expr -> expr {print("+")} expr + | expr {print("-")} expr -  | {print("(")} expr {print(")\*(")} expr {print(")")} \*  | {print("(")} expr {print(")/(")} expr {print(")")} / | digit {print(digit)} |

### Another reference answer

|  |
| --- |
| E -> {print("(")} E {print(op)} E {print(")"}} op | digit {print(digit)} |

## 2.3.c

Construct a syntax-directed translation scheme that translates integers into roman numerals.

### Answer

assistant function:

|  |
| --- |
| repeat(sign, times) // repeat('a',2) = 'aa' |

translation schemes:

|  |
| --- |
| num -> thousand hundred ten digit  { num.roman = thousand.roman || hundred.roman || ten.roman || digit.roman; print(num.roman)}  thousand -> low {thousand.roman = repeat('M', low.v)} hundred -> low {hundred.roman = repeat('C', low.v)}  | 4 {hundred.roman = 'CD'}  | high {hundred.roman = 'D' || repeat('X', high.v - 5)}  | 9 {hundred.roman = 'CM'}  ten -> low {ten.roman = repeat('X', low.v)}  | 4 {ten.roman = 'XL'}  | high {ten.roman = 'L' || repeat('X', high.v - 5)}  | 9 {ten.roman = 'XC'}  digit -> low {digit.roman = repeat('I', low.v)}  | 4 {digit.roman = 'IV'}  | high {digit.roman = 'V' || repeat('I', high.v - 5)}  | 9 {digit.roman = 'IX'} low -> 0 {low.v = 0} | 1 {low.v = 1}  | 2 {low.v = 2} | 3 {low.v = 3} high -> 5 {high.v = 5} | 6 {high.v = 6}  | 7 {high.v = 7}  | 8 {high.v = 8} |

|  |  |
| --- | --- |
| **2.3.d**  Construct a syntax-directed translation scheme that trans lates roman numerals into integers.  **Answer**  productions:  romanNum -> thousand hundred ten digit thousand -> M | MM | MMM | ε  hundred -> smallHundred | C D | D smallHundred | C M smallHundred -> C | CC | CCC | ε ten -> smallTen | X L | L smallTen | X C  smallTen -> X | XX | XXX | ε  digit -> smallDigit | I V | V smallDigit | I X  smallDigit -> I | II | III | ε translation schemes:  romanNum -> thousand hundred ten digit {romanNum.v = thousand.v || hundred.v || ten.v  || digit.v; print(romanNun.v)} thousand -> M {thousand.v = 1} | MM {thousand.v = 2}  | MMM {thousand.v = 3} | ε {thousand.v = 0}  hundred -> smallHundred {hundred.v = smallHundred.v}  | C D {hundred.v = smallHundred.v}  | D smallHundred {hundred.v = 5 + smallHundred.v}  | C M {hundred.v = 9} smallHundred -> C {smallHundred.v = 1} | CC {smallHundred.v = 2}  | CCC {smallHundred.v = 3} | ε {hundred.v = 0} ten -> smallTen {ten.v = smallTen.v}  | X L {ten.v = 4}  | L smallTen {ten.v = 5 + smallTen.v}  | X C {ten.v = 9} smallTen -> X {smallTen.v = 1} | XX {smallTen.v = 2}  | XXX {smallTen.v = 3} | ε {smallTen.v = 0}  digit -> smallDigit {digit.v = smallDigit.v}  | I V {digit.v = 4}  | V smallDigit {digit.v = 5 + smallDigit.v}  | I X {digit.v = 9} smallDigit -> I {smallDigit.v = 1} | II {smallDigit.v = 2}  | III {smallDigit.v = 3}  | ε {smallDigit.v = 0} | |
| **2.3.e**  Construct a syntax-directed translation scheme that translates postfix arithmetic expressions into equivalent prefix arithmetic expressions.  **Answer**  production:   |  | | --- | | expr -> expr expr op | digit |   translation scheme:   |  | | --- | | expr -> {print(op)} expr expr op | digit {print(digit)} |     **Exercises for Section 2.4**  **2.4.a**  Construct recursive-descent parsers, starting with the following grammars:   1. S -> + S S | - S S | a 2. S -> S ( S ) S | ε 3. S -> 0 S 1 | 0 1   **Answer**  See [2.4.1.1.c,](http://dragon-book.jcf94.com/book/ch02/2.4/2.4.1.1.c) [2.4.1.2.c,](http://dragon-book.jcf94.com/book/ch02/2.4/2.4.1.2.c) and [2.4.1.3.c](http://dragon-book.jcf94.com/book/ch02/2.4/2.4.1.3.c) for real implementations in C.   |  | | --- | | lookahead = nextTerminal();  }else{  throw new SyntaxException()  }  } |   1） S -> + S S | - S S | a   |  | | --- | | void S(){  switch(lookahead){ case "+": match("+"); S(); S(); break; case "-": match("-"); S(); S(); break; case "a": match("a"); break; default:  throw new SyntaxException();  } } void match(Terminal t){ if(lookahead = t){ |  1. S -> S ( S ) S | ε  |  | | --- | | void S(){  if(lookahead == "("){  match("("); S(); match(")"); S();  }  } |  1. S -> 0 S 1 | 0 1  |  | | --- | | void S(){  switch(lookahead){ case "0": match("0"); S(); match("1"); break; case "1":  // match(epsilon); break; default: throw new SyntaxException();  }  } | |