CS 316 E6TBA Queens College CUNY
Exercise Set #1

1. Compare virtual machines and hardware machines for a programming language *L* and point out one advantage of each.

2. Suppose that a high-level language *H* is compiled into an intermediate language *I*, which in turn is compiled into a native machine language *M*. Each of these can be executed by a virtual machine or a hardware machine. Then there are six possible execution methods for the high-level language *H*:

```
VH: virtual machine for H
HH: hardware machine for H
VI: virtual machine for I
HI: hardware machine for I
VM: virtual machine for M
HM: hardware machine for M
```

1. Which execution method is the slowest of the six?
2. Which execution method is the fastest of the six?
3. Draw a lattice of these six methods, with links "X → Y" meaning that method Y is faster than (or more or less equal in speed to) method X.

3. Concisely explain the difference(s) between intermediate-code compilers and native-code compilers with regard to how high-level source code is compiled and executed.

4. Concisely explain how the use of a well-designed intermediate language would facilitate construction of compilers that translate *m* high-level languages into *n* native machine languages.

5. Give three main components of the Java Virtual Machine and succinctly describe the function of each.

6. Give a flow diagram specifying the input to and output from each of the following stages of compilers. If error messages are possible output, indicate them appropriately as well.

1. lexical analysis
2. syntactic analysis
3. type checking
4. semantic analysis and intermediate code generation
5. optimization
6. object code generation

7. Give a complete BNF grammar for each of the following. You may use extended BNF.

1. *Identifiers* : Any letter followed by zero or more letters or digits.
2. *Signed Integers* : "+" or "−" followed by one or more digits.

8. Consider the following extended BNF grammar (EBNF grammar):

<id > → <letter > { <letter > | <digit > }
<extended id > → <id > { "_" <letters and digits > }
<letters and digits > → { <letter > | <digit > }$^{+}$

1. Determine if each of the following strings belongs to the category <extended id >:

    8ABC CS316 CS316_ CS316_ABCX
    _CS316 CS316_ _ABC CS316_987 CS316_ABC_32A
    CS316_543_7B5 CS316_A_

2. Rewrite the above grammar to an equivalent one in the original BNF without using any notation in EBNF.

9. Given is the following EBNF grammar:

< float > → [+ | −] {< digit >} "." {< digit >}$^{+}$ [(E | e) [+ | −] {< digit >}$^{+}$]

Determine if each of the following strings belongs to the category <float >:

.e1 .2e .2e3 .45E−32 3.2145e10 768.43 2709 2709.
−.562e2 +34E+5 −65.67 75647.74653e− 756.65e−7564 +.64−8

In the following, presume that the syntactic category <id > is always defined by:

<id > → <letter > { <letter > | <digit > }

10. Consider the following EBNF grammar:

< sequence > → "(" < elements > ")"
< elements > → < element > { "," < element > }
< element > → < id > | < sequence >

1. Determine if each of the following strings belongs to the category <sequence>; if it does, give a parse tree for it.

    ( )
    (xyz)
    (x, y, z)
    ((x, y), z, (y))
    (x) (y)
    ((x, y, z), ((x, y), (z)))
    x) y) z

2. Rewrite the production rule for <elements> to an equivalent one in the original BNF without using any notation in EBNF.

11. Consider the following EBNF grammar:

< X > → "{" < inside > "}" | < id > ";"

< inside > → { < X > }$^{+}$

Determine if each of the following strings belongs to the category < X >; if it does, give a parse tree for it.

{ a; b; c; }
{ a   b {
{ a; { b; c; } }
} a; b; {
{ { a; } b; { c; d; } }

12. Consider the following EBNF grammar:

< E > → (+|−) < E > < E > | < id >

This is known as expressions in *prefix form*, since the binary operators + and − come before two arguments. This grammar is known to be unambiguous (and hence does not need disambiguation parentheses). Determine if each of the following strings belongs to the category < E >; if it does, give a parse tree for it.

abc
+   abc   xyz
+   +   abc   xyz   ABC
−   abc   +   xyz
−   abc   +   abc   xyz

"abc", "xyz", "ABC" are identifiers.

13. Consider the following EBNF grammar:

< E > → < E > < E > (+|−) | < id >

This is known as expressions in *postfix form*, since the binary operators + and − come after two arguments. This grammar is known to be unambiguous (and hence does not need disambiguation parentheses). Determine if each of the following strings belongs to the category < E >; if it does, give a parse tree for it.

abc
abc   xyz   +
abc   xyz   +   ABC   +
abc   −   xyz   +
abc   abc   xyz   +   −

"abc", "xyz", "ABC" are identifiers.

14. Consider the following BNF grammar for arithmetic expressions, which incorporates the *unary* "−" operator:

< E > → < term > | < term > + < E >
< term > → < primary > | < primary > * < term >
< primary > → < id > | "(" < E > ")" | − < primary >

1. Give the parse tree for each of the following:

   x + y * z
   (x + y) * z
   x + y + z
   x * y * z
   x + − y * − z
   − (x + y) * z

2. According to this grammar, are the + and * operators left-associative or right-associative, which?
3. Expand this grammar to incorporate the binary subtraction and division operators "–" and "/" with their proper precedence.

15. Consider the following BNF grammar for Boolean expressions:

<BE > → <id > | <BE > "||" <BE > | <BE > "&&" <BE > | "!" <BE > | "(" <BE > ")"

1. Give *all* parse trees for: ! x || y
2. Give *all* parse trees for: x || y && x || z
3. This grammar is ambiguous – explain why.
4. Give an unambiguous BNF grammar which equivalently defines < BE > by incorporating the operator precedence rules: "!" has higher precedence than "&&", which has higher precedence than "||". You may introduce auxiliary syntactic categories and use EBNF.

16. Consider the following BNF grammar for conditionals:

< cond > → **if** "(" < B > ")" < S > | **if** "(" < B > ")" < S > **else** < S >
< S > → < cond > | .... other kinds of statements ....

1. Give *all* parse trees for: if (B_1) if (B_2) S_1 else S_2 . Assume that B_1 and B_2 are some Boolean expressions and S_1 and S_2 are some statements – derive these directly from < B > and < S >.
2. This grammar is ambiguous – explain why.

17. Consider the unambiguous BNF grammar for the conditional statements using < matched S > and < unmatched S > discussed in class. Using this grammar, give the parse tree for:

1. if (B_1) if (B_2) if (B_3) S_1 else S_2 else S_3
2. if (B_1) S_1 else if (B_2) S_2 else S_3

B_1, B_2, B_3 are Boolean expressions and S_1, S_2, S_3 are matched statements – derive these directly from < B > and < matched S >.

18. Concisely explain what problem will arise in compiler construction if an ambiguous BNF grammar is used to describe programming language syntax.