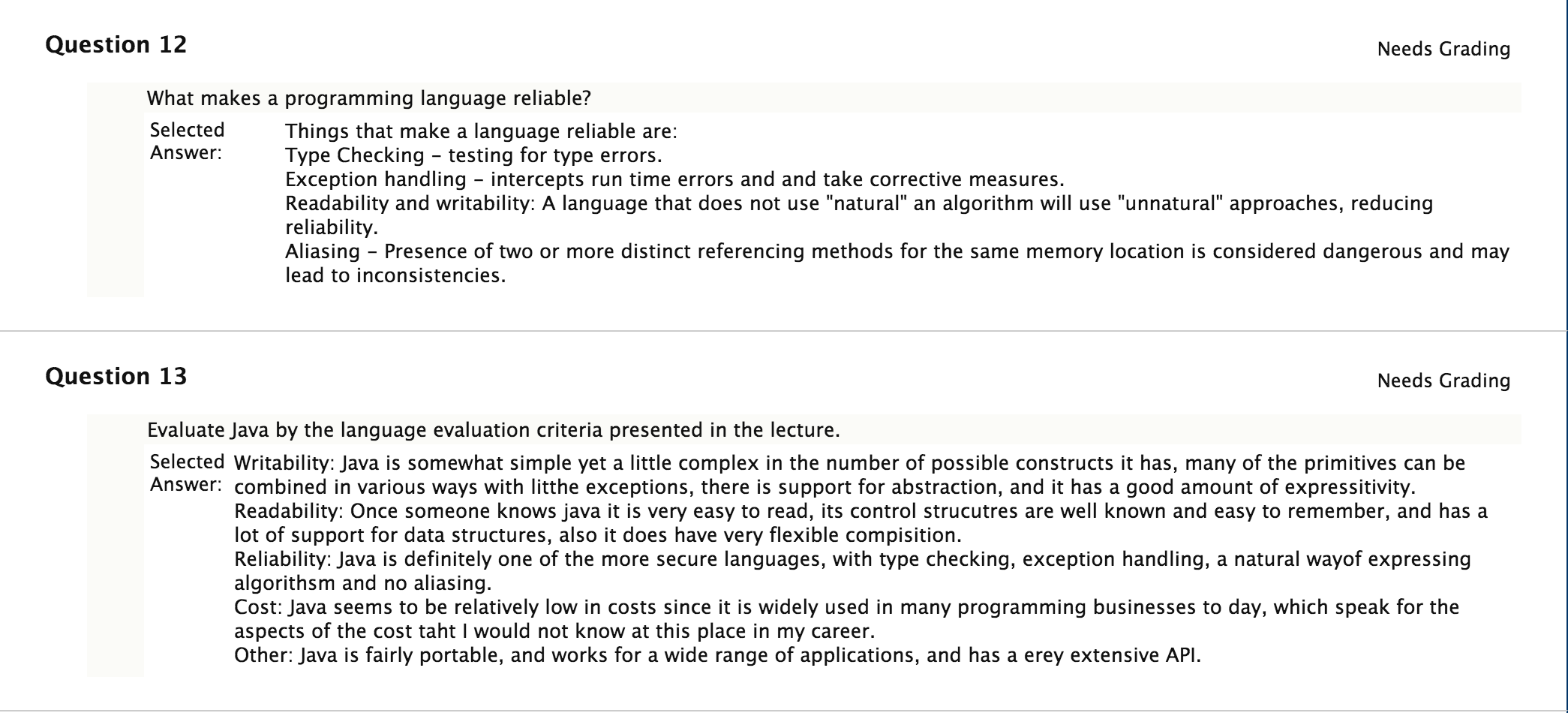
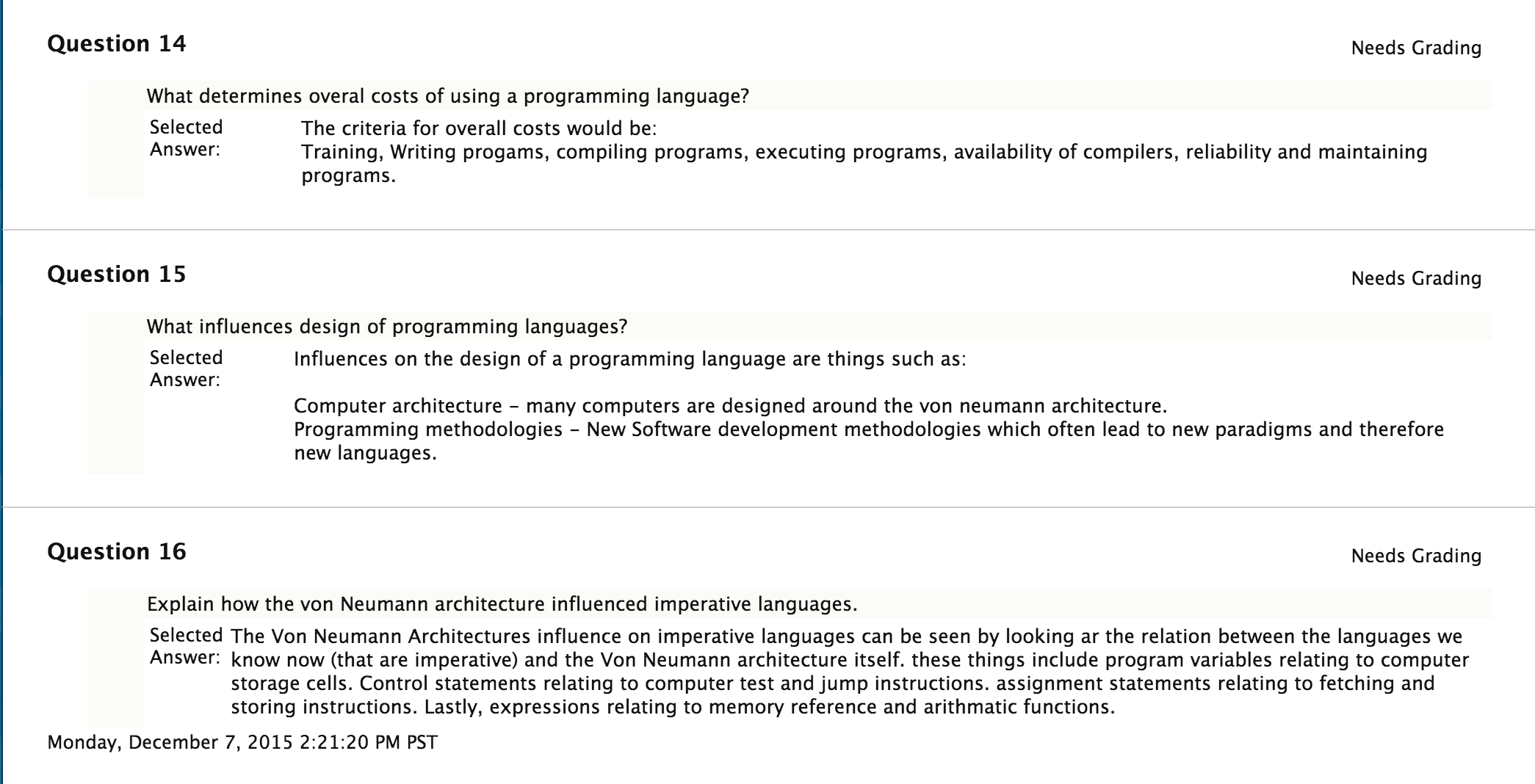
SET 1

Overview of Programming Languages

|  |  |
| --- | --- |
| Screen Shot 2015-12-07 at 11.52.15 AM.png  Screen Shot 2015-12-07 at 11.53.05 AM.png | |
| Screen Shot 2015-12-07 at 2.23.34 PM.png  Screen Shot 2015-12-07 at 2.21.30 PM.png | |
|  | |





SET 2

Corey: So i guess Kevin never got around to grading this one.

SET 3

Language Specifications

Question 2:

Extend the following sample grammar shown in the class:

<program> ::= <statement> | <program> <statement>

<statement> ::= <assignStmt> | <ifStmt>

<assignStmt>::= <id> = <expr> ;

<ifStmt> ::= if ( <expr> ) <stmt> ;

<expr> ::= <id> | <int> | <expr> + <expr>

<id> ::= a | b | c | i | j | k | n | x | y | z

<int> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

so the following program can be parsed:

i = 1; a = 3;

while ( i > 0 )

{ i = i - 1; s = s + a; }

Show:

2.1) the new grammar

2.2) the rightmost derivation of the program

=========================================

2.1)

<program> ::= <statement> | <program> <statement>

<statement> ::= <assignStmt> | <ifStmt> | <whileStmt>

<assignStmt>::= <id> = <expr> ;

<ifStmt> ::= if ( <expr> ) <stmt> ;

<whileStmt> ::= while ( <expr> {< | > } <expr> ) { <program> } ;

<ifStmt> ::= if ( <expr> ) <stmt> ;

<expr> ::= <id> | <int> | <expr> {+|-} <expr>

<id> ::= a | b | c | i | j | k | n | s | x | y | z

<int> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

2.2) Rightmost Derivation

<program>

--> <program> <statement>

--> <program> <whileStmt>

--> <program> while(<expr> > <expr>) <program>;

-/-> <program> while(<expr> > <expr>) <program> <statement>;

--> <program> while(<expr> > <expr>) <program> <assignStmt>

--> <program> while(<expr> > <expr>) <program> <id> = <expr>;

--> <program> while(<expr> > <expr>) <program> <id> = <expr> + <expr>;

--> <program> while(<expr> > <expr>) <program> <id> = <expr> + <id>;

--> <program> while(<expr> > <expr>) <program> <id> = <expr> + a;

--> <program> while(<expr> > <expr>) <program> <id> = <id> + a;

--> <program> while(<expr> > <expr>) <program> <id> = s + a;

--> <program> while(<expr> > <expr>) <program> s = s + a;

--> <program> while(<expr> > <expr>) <statement> s = s + a;

--> <program> while(<expr> > <expr>) <assignStmt> s = s + a;

--> <program> while(<expr> > <expr>) <id> = <expr>; s = s + a;

--> <program> while(<expr> > <expr>) <id> = <expr> - <expr>; s = s + a;

--> <program> while(<expr> > <expr>) <id> = <expr> - <int>; s = s + a;

--> <program> while(<expr> > <expr>) <id> = <expr> - 1; s = s + a;

--> <program> while(<expr> > <expr>) <id> = <id> - 1; s = s + a;

--> <program> while(<expr> > <expr>) <id> = i - 1; s = s + a;

--> <program> while(<expr> > <expr>) i = i - 1; s = s + a;

--> <program> while(<expr> > <int>) i = i - 1; s = s + a;

--> <program> while(<expr> > 0) i = i - 1; s = s + a;

--> <program> while(<id> > 0) i = i - 1; s = s + a;

--> <program> while(i > 0) i = i - 1; s = s + a;

--> <program> <statement> while(i > 0) i = i - 1; s = s + a;

--> <program> <assignStmt> while(i > 0) i = i - 1; s = s + a;

--> <program> <id> = <expr>; while(i > 0) i = i - 1; s = s + a;

--> <program> <id> = <int>; while(i > 0) i = i - 1; s = s + a;

--> <program> <id> = 3; while(i > 0) i = i - 1; s = s + a;

--> <program> a = 3; while(i > 0) i = i - 1; s = s + a;

--> <statement> a = 3; while(i > 0) i = i - 1; s = s + a;

--> <assignStmt> a = 3; while(i > 0) i = i - 1; s = s + a;

--> <id> = <expr>; a = 3; while(i > 0) i = i - 1; s = s + a;

--> <id> = <int>; a = 3; while(i > 0) i = i - 1; s = s + a;

--> <id> = 1; a = 3; while(i > 0) i = i - 1; s = s + a;

--> i = 1; a = 3; while(i > 0) i = i - 1; s = s + a;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Question 3:

i = 1; a = 3;

while ( i > 0 )

{ i = i - 1; s = s + a; }

3.1) Same as 2.1)

<program> ::= <statement> | <program> <statement>

s<statement> ::= <assignStmt> | <ifStmt> | <whileStmt>

<assignStmt>::= <id> = <expr> ;

<ifStmt> ::= if ( <expr> ) <stmt> ;

<whileStmt> ::= while ( <expr> {< | > } <expr> ) <program> ;

<ifStmt> ::= if ( <expr> ) <stmt> ;

<expr> ::= <id> | <int> | <expr> {+|-} <expr>

<id> ::= a | b | c | i | j | k | n | s | x | y | z

<int> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

3.2) Show Leftmost derivation

<program>

--> <program> <statement>

--> <program> <statement> <statement>

--> <statement> <statement> <statement>

--> <assignStmt> <statement> <statement>

--> <id> = <expr>; <statement> <statement>

--> i = <expr>; <statement> <statement>

--> i = <int>; <statement> <statement>

--> i = 1; <statement> <statement>

--> i = 1; <assignStmt> <statement>

--> i = 1; <id> = <expr>; <statement>

--> i = 1; a = <expr>; <statement>

--> i = 1; a = <int>; <statement>

--> i = 1; a = 3; <statement>

--> i = 1; a = 3; <whileStmt>

--> i = 1; a = 3; while(<expr> > <expr>) <program>;

--> i = 1; a = 3; while(<id> > <expr>) <program>;

--> i = 1; a = 3; while(i > <expr>) <program>;

--> i = 1; a = 3; while(i > <int>) <program>;

--> i = 1; a = 3; while(i > 0) <program>;

--> i = 1; a = 3; while(i > 0) <program> <statement>;

--> i = 1; a = 3; while(i > 0) <statement> <statement>;

--> i = 1; a = 3; while(i > 0) <assignStmt> <statement>;

--> i = 1; a = 3; while(i > 0) <id> = <expr>; <statement>;

--> i = 1; a = 3; while(i > 0) i = <expr>; <statement>;

--> i = 1; a = 3; while(i > 0) i = <expr> - <expr>; <statement>;

--> i = 1; a = 3; while(i > 0) i = <expr> - <expr>; <statement>;

--> i = 1; a = 3; while(i > 0) i = i - <expr>; <statement>;

--> i = 1; a = 3; while(i > 0) i = i - <int>; <statement>;

--> i = 1; a = 3; while(i > 0) i = 1 - 1; <statement>;

--> i = 1; a = 3; while(i > 0) i = 1 - 1; <assignStmt>;

--> i = 1; a = 3; while(i > 0) i = 1 - 1; <id> = <expr>;

--> i = 1; a = 3; while(i > 0) i = 1 - 1; s = <expr>;

--> i = 1; a = 3; while(i > 0) i = 1 - 1; s = <expr> + <expr>;

--> i = 1; a = 3; while(i > 0) i = 1 - 1; s = <id> + <expr>;

--> i = 1; a = 3; while(i > 0) i = 1 - 1; s = s + <expr>;

--> i = 1; a = 3; while(i > 0) i = 1 - 1; s = s + <id>;

--> i = 1; a = 3; while(i > 0) i = 1 - 1; s = s + a;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Question 4

Consider the following formal grammar G = (N, Σ, P, S):

N={S}

Σ={a, b, c}

P={1: S→aSbSa, 2: S→a, 3: S→b}

Show 10 words that belong to the language L(G) along with their derivations.

Is the word aabbaababbaa in the language? If yes, show the derivation.

What about aaabbabbbaaabbbbabbbb? Show the derivation if it exists.

Is this a context-free grammar (CFG)? Justify.

============================

4.1)

1) S→a → (2) a

2) S→b → (3) b

3) S→aSbSa → (2) aabSa → (2) aabaa

4) S→aSbSa → (2) aabSa → (3) aabba

5) S→aSbSa → (3) abbSa → (2) abbaa

6) S→aSbSa → (3) abbSa → (3) abbba

7) S→aSbSa → (1) aaSbSabSa → (2) aaabSabSa → (2) aaabaabSa → (2) aaabaabaa

8) S→aSbSa → (1) aaSbSabSa → (2) aaabSabSa → (2) aaabaabSa → (3) aaabaabba

9) S→aSbSa → (1) aaSbSabSa → (3) aabbSabSa → (2) aaabaabSa → (2) aaabaabaa

10) S→aSbSa → (1) aaSbSabSa → (3) aabbSabSa → (3) aabbbabSa → (2) aabbbabaa

4.2)

aabbaababbaa is not in the language because after 2 productions with rule 1, the result is 13 characters when the string is only 12 characters long.

4.3)

aaabbabbbaaabbbbabbbb is not in the language. The leftmost derivation leads to a string with an S non-terminal on the right side, allowing us to prematurely end the derivation with only 2 b's at the end. We can also use rule 1 but that would leave the string with an a at the end.

4.4)

This is a context-free grammar because G can be represented as G = (N, Σ, P, S) and each production has a single non-terminal symbol on the left-hand side.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Question 5

Show that the following grammar is ambiguous.

Hint: Show two different leftmost or rightmost derivations for the same string. Equivalently, you can show two different parse trees for the same string.

<expr> ::= <expr> + <expr> | <expr> - <expr> | <expr> \* <expr> | <expr> / <expr> | int

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

Using this grammar show that ambiguity is not acceptable for programming languages.

Hint: Show that the same expression can be evaluated in different ways depending on the derivation.

============================

1 + 2 - 3

Left

<expr>

--> <expr> + <expr>

--> 1 + \*<expr>\*

--> 1 + <expr> - <expr>

--> 1 + 2 - <expr>

--> 1 + 2 - 3

Right

--> \*<expr>\* - <expr>

--> <expr> + <expr> - <expr>

--> 1 + <expr> - <expr>

--> 1 + 2 - <expr>

--> 1 + 2 - 3

=======

Question 6

Describe the languages generated by the regular expressions from the following table:

[abc]+

[abc]\*

[0-9]+

[1-9][0-9]\*

[a-zA-Z][a-zA-Z0-9\_]\*

(19|20)[0-9]{2}[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])

1) Strings with one or more of a, b, or c (exclusively - no mixing of a, b, c).

2) Strings with zero or more of a, b, or c (exclusively - no mixing of a, b, c).

3) Strings with one or more of the digits between 0-9.

4) Strings beginning with a digit between 1-9 and ending with zero or more digits between 0-9.

5) Strings beginning with an upper or lower case letter and ending with zero or more alphanumeric characters.

6) String beginning with 19 or 20,

with 2 digits between 0-9 following,

with either a -, /, or . following,

and either a 0 followed by a digit between 1-9

OR

a 1 followed by 0, 1, or 2,

followed by either a -, /, or .,

and finally one of these following 3 cases:

0 followed by a digit 1-9

either 1 or 2 followed by a digit 0-9

3 followed by 0 or 1.

SET 4

Overview of Program Compilation

Question 1:

Describe what Syntax Analysis is and what it entails. Contrast it with Semantic Analysis.

===

Syntax Analysis involves processing source code and tokenizing the input. Syntax analysis only tokenizes the input with no regard to the actual meaning of it.

Semantic Analysis, by contrast, involves adding understanding to the source code by doing type checking, checking that the program obeys language rules such as "declare before use," and checking type compatibility, among other things.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Question 2:

Write a comparative analysis of compilers vs. interpreters.

===

In compilation, a compiler reads and analyzes an entire source program from one language (source language) and translates it to another language (the target language). It allows an opportunity for optimization and better performance. When the program executes, the instructions run on the target machine's native code.

In interpretation, the interpreter is the "locus of control" during program execution by reading the source code and producing the results. The program execution is interleaved with analysis, therefore in general, interpreted programs are slower than compiled programs. However, interpretation allows greater flexibility through better diagnostics (error messages) and easier debugging.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Question 3:

Contrast language preprocessors and compilers.

===

The biggest difference between preprocessors and compilers is that a preprocessor will produce an Intermediate Representation (IR) of a program while a compiler will produce object code. A language preprocessor simply removes comments/whitespace and expands abbreviations. A preprocessor may also identify higher-level syntactic structures (eg loops) and may group characters into tokens (keywords, identifiers, numbers, symbols -- lexical preprocessing). However, a preprocessor will often let errors through. A preprocessor produces an IR which the compiler will take as input, and will then use it to generate object code.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Question 4:

Explain what a just-in-time compilation is and how it improves efficiency of interpreted programming languages.

===

Just-in-Time compilation allows the compiler to generate code that makes assumptions that are finalized on runtime. Only after the program is executed will the JIT compiler compile the source program. Some of the optimizations that JIT can perform are maintaining a call count of methods, and if a method is called enough times, it will be compiled as soon as the program is executed, whereas other methods may not be compiled at all. The Java JIT compiler, in particular, after recompiling a method, will try to apply more optimizations that the previous compilation.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Question 5:

Describe all stages necessary to run a program starting with the source code. If there are different options, describe each of them.

===

bad:

First, the source program is analyzed by the preprocessor or scanner which removes comments/whitespace and expands abbreviations in the style of a macro assembler. The preprocessor may also identify high-level syntactic structures such as loops. This intermediate form is then passed to the parser, and the parser reads the token stream and generates an Intermediate Representation (also called an IR, or a parse tree). Next, the abstract parse tree undergoes semantic analysis where an actual understanding of the code takes place; type checking, adherence to language rules, and other tasks are executed in order to prevent errors from making it to the next stage. The output is a Decorated AST. Finally, intermediate code generation occurs followed by optional optimizations.

---

good:

First, the source program undergoes preprocessing. The scanner removes whitespace/comments and expands abbreviations (such as #define in C). The modified source program then undergoes semantic analysis where an actual understanding of the code takes place; type checking, adherence to language rules, and other tasks are executed in order to prevent errors from making it to the next stage. Most errors and warnings are produced during this stage. If an optimization flag was specified, the object code is then optimized. Then the linker merges any required methods from libraries into the object code.

Preprocessing is the first stage and it has 2 components. The first is scanning (lexical analysis) where the source program is tokenized. The next stage is parsing (syntactic analysis) where the tokens are combined to form a parse tree.

Next, semantic analysis occurs and it is where an actual understanding of the code takes place; type checking, obeyance of language rules, and other tasks are executed in order to prevent errors from making it to the next stage. This stage will commonly occur after parsing, and the preprocessor will produce an Intermediate Representation (IR) of the source program.

The IR is then received by the compiler which produces object code. If any libraries were used, the methods needed are integrated into the object code by the linker.

Finally, the linker links the object files and produces an executable.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Question 6:

Describe the purpose, functionality, and constraints of a compiler front end.

===

A compiler front end deals with analysis and is composed of 2 parts; the scanner and the parser. The front end must analyze a source program and understand its structure and meaning. It must recognize a programs' legality, manage the storage of all variable, and it maps the source program to an Intermediate Representation (often a parse tree, which is produced by the parser). However, the front end only goes so far as to understand a program and does not actually generate a program in the target language.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Question 7:

Describe the purpose, functionality, and constraints of a compiler back end.

===

A compiler backend deals with synthesis and generates the equivalent target language program by translating the IR to target machine code. The output of the backend must agree with the operating system and linker on the target format, and it should produce fast, compact code. It should use machine resources, such as registers, instructions, and the memory hierarchy, effectively.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Question 8:

Describe what a scanner is. Described its purpose and how it works.

Give some examples.

===

The purpose of a scanner is to preprocess a source program. It removes whitespace/comments and tokenizes the source program, sometimes identifying high-level syntactic structures such as loops. However, the scanner will not catch errors. A scanner produces tokens which are passed to the parser.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Question 9:

Describe what a parser is. Describe its purpose and how it works.

Give some examples.

===

A parser actually "understands" a source program and during the parsing process, a parser may enforce type compatibility and obeyance of language rules (such as declare before us). A parser produces an Intermediate Representation, often a parse tree, of the program which contains the meaning of a program without syntactic noise.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SET 5

Lexical Analysis

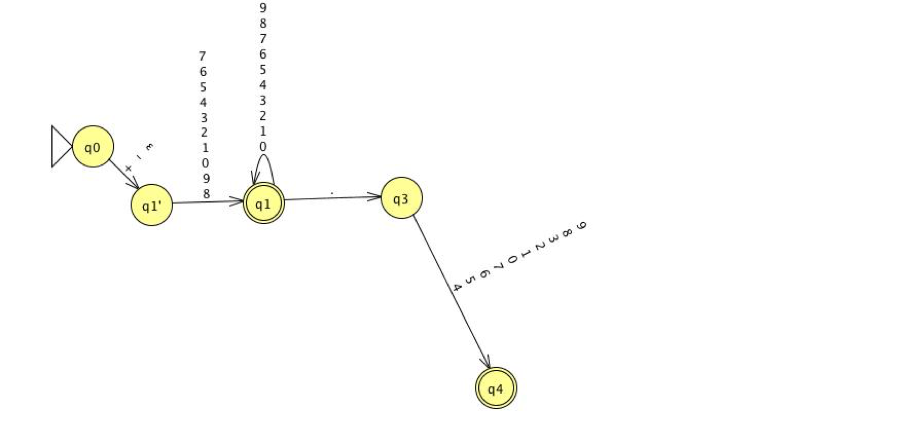
**Question 1**

Create a regular expression that defines a float number (with possible +/- in front of digits with a dot or without).

Using qfsm create an FSM that recognizes the number.

Use the simulator to verify that your automaton works as expected (i.e., recognizes float numbers).

Regex: (+∪-∪ɛ) (.(0∪1∪…∪8∪9 )+∪((0∪1∪…∪8∪9 )+.(0∪1∪…∪8∪9)+) )

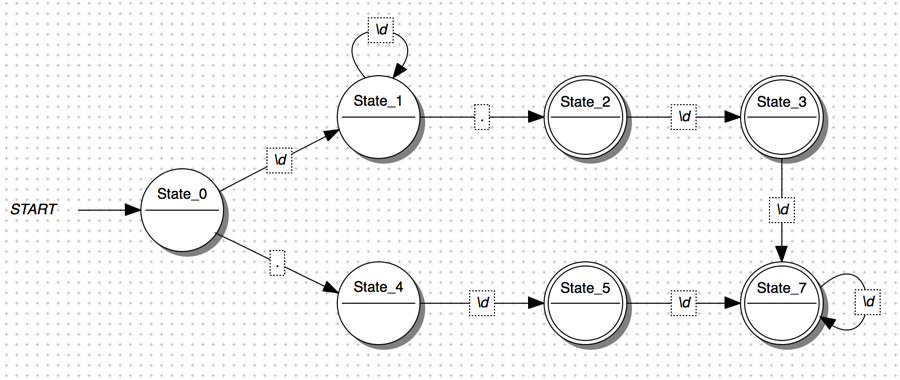


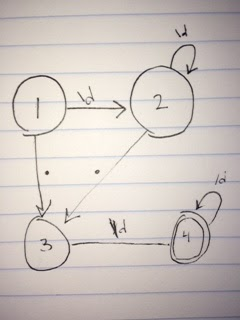
### **Question 2**

Convert step by step the following NFA to a minimal Deterministic Finite Automaton (DFA):

Show all the stages of the conversion using qfsm (use ASCII mode).

Vreify that your DFA accepts exactly same strings as the NFA.





### **Question 3**

Consider the following grammar for a simple programming language:

<program> ::= <statement> | <program> <statement>

<statement> ::= <assignStmt> | <repeatStmt> | <printStmt>

<assignStmt> ::= <id> = <expr> ;

<repeatStmt> ::= repeat ( <expr> ) <stmt>

<printStmt> ::= print <expr> ;

<expr> ::= <term> | <expr> <addOp> <term>

<term> ::= <factor> | <term> <multOp> <factor>

<factor> ::= <id> | <number> | - <factor> | ( <expr> )

<id> ::= <letter> | <id> <letter>

<letter> ::= a | b | c | d | e | f | g | h | i | j

| k | l | m | n | o | p | r | s | t

| u | v | w | x | y | z

<number> ::= <digit> | <number> <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<addOp> ::= + | -

<multOp> ::= \* | / | %

Create a state transition table for the scanner of this language using an adjacency table. Assign a number to each of the input symbols, and a number to each of the states. Use a comma as a separator. Remember that you need to have the same number of commas in each row, just put a space between the commas for an empty table cell.

Also submit C data structures to implement the transition table.

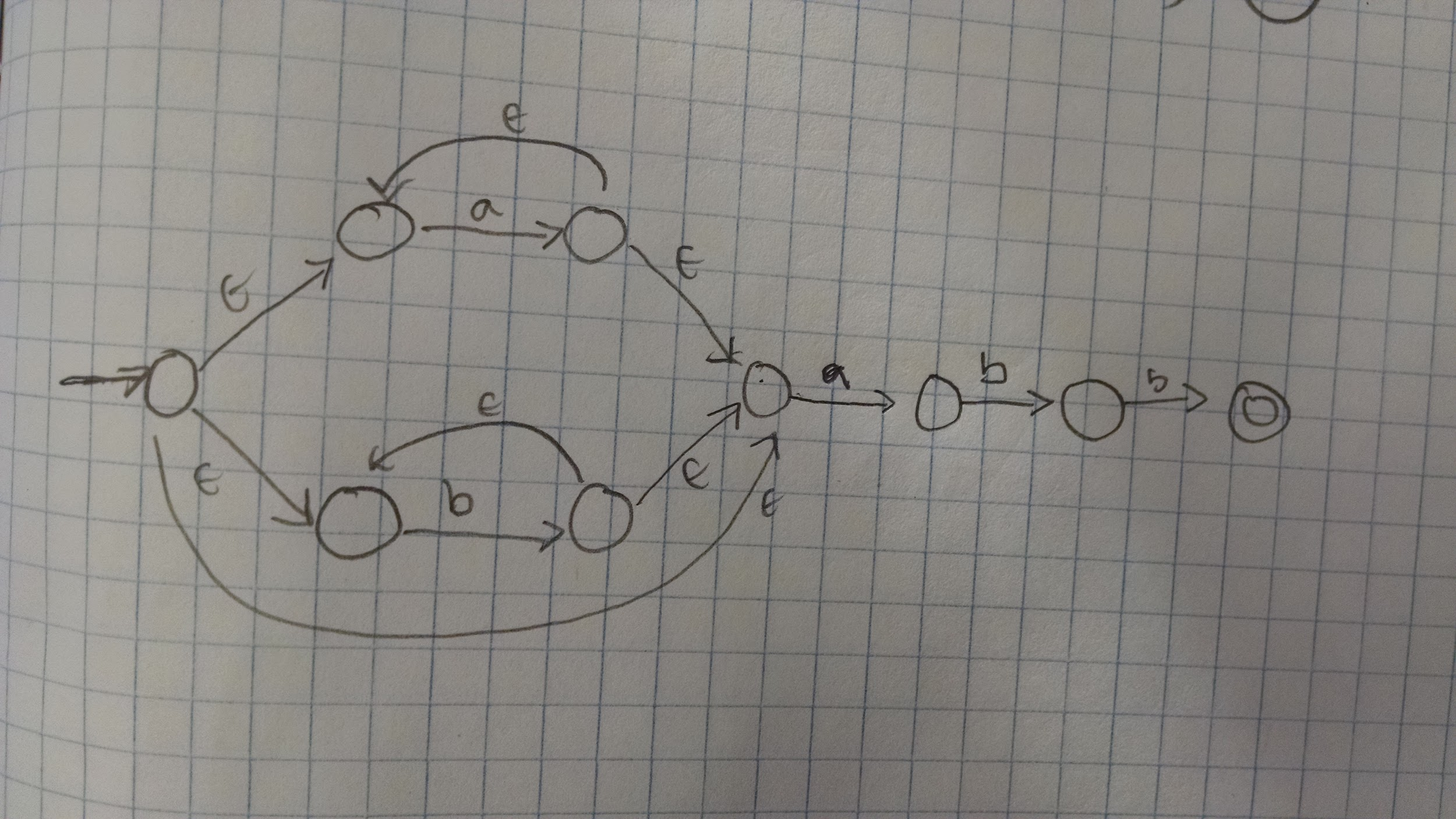
### **Question 4**

Convert step by step the following regular expression to an NFA (non-deterministic finite automaton):

((a|b)\*abb)

Use qfsm (ASCII mode) to show all the stages of the conversion (i.e., create multiple FSMs of all components that you construct as you progress through the construction).

At each stage, run the simulator to verify that your FSM works as expected.



### **Question 5**

Consider the following grammar for a simple programming language:

<program> ::= <statement> | <program> <statement>

<statement> ::= <assignStmt> | <repeatStmt> | <printStmt>

<assignStmt> ::= <id> = <expr> ;

<repeatStmt> ::= repeat ( <expr> ) <stmt>

<printStmt> ::= print <expr> ;

<expr> ::= <term> | <expr> <addOp> <term>

<term> ::= <factor> | <term> <multOp> <factor>

<factor> ::= <id> | <number> | - <factor> | ( <expr> )

<id> ::= <letter> | <id> <letter>

<letter> ::= a | b | c | d | e | f | g | h | i | j

| k | l | m | n | o | p | r | s | t

| u | v | w | x | y | z

<number> ::= <digit> | <number> <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<addOp> ::= + | -

<multOp> ::= \* | / | %

Using qfsm (ASCII mode), create a DFA that will recognize all tokens in this grammar.

Verify the correctness using the simulator.

**{TODO}**

SET 6

Parsing

### **Question 1**

Using the LL(1) grammar along with the transition table from the lecture:

(1) S → F

(2) S → ( S + F )

(3) F → 1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x | ( | ) | 1 | + | $ |
| S | 2 |  | 1 |  |  |
| F |  |  | 3 |  |  |

parse the following string: ((1+1)+1)

Show the stack and the input buffer for all stages. Use the following format: [stack][input][transition]

For example, for (1+1):

[$][(1+1)][]

[S$][(1+1)][]

[(S+F)$)][(1+1)][2]

[S+F)$][1+1)][]

[F+F)$][1+1)][1]

[1+F)$][1+1)][3]

[+F)$][+1)][]

[F)$][1)][]

[1)$][1)][3]

[)$][)][]

[$][][]

[$][((1+1)+1)][] // Push S into stack for empty string (initial read)

[S$][((1+1)+1)][] // We can address table[S][‘(‘] which gives Rule 2

[(S+F)$] [((1+1)+1)][2] // Read ‘(‘ from input, use Rule 2 to replace S with (S+F)

[S+F)$] [(1+1)+1)][] // Read ‘(‘ in stack and input, so pop both

[S+F)$] [(1+1)+1)][] // table[S][‘(‘] gives Rule 2, replace S with S+F

[(S+F)+F$][(1+1)+1][2] // Read ‘(‘ from stack and ‘(‘ from input, pop both

[S+F)+F$][1+1)+1][] // table[S][1] gives Rule 1 (replace S with F)

[F+F)+F$][1+1)+1][1] // table[F][1] gives Rule 3 (replace F with 1)

[1+F)+F$][1+1)+1][3] // Have 1 in stack and 1 in input, pop both

[+F)+F$][+1)+1][] // Have + in stack and + and input, pop both

[F)+F$][1)+1][] // table[F][1] gives Rule 3 (replace F with 1)

[1)+F$][1)+1][3] // Read 1 in stack and 1 in input, pop both

[)+F$][)+1][] // Read ‘)’ in stack and in input, pop both

[+F$][+1][] // Read ‘+’ in stack and input, pop both

[F$][1][] // table[F][1] gives Rule 3 (replace F with 1)

[1$][1][] // Read ‘1’ in stack and ‘1’ in input, pop both

[$][][] // Finished

### **Question 2**

Using the LR(1) grammar:

(1) E → E \* B

(2) E → E + B

(3) E → B

(4) B → 0

(5) B → 1

and associated transition table from the lecture example (attached as an image), parse the following string:

1+0\*1+0

Use the following notation:

[state][stack][transition]

For example, for the string: 1+1

the following sequence would describe the parsing:

[s0][1][]

[s2,s0][B][5]

[s0][B][]

[s4,s0][E][3]

[s0][E][]

[s3,s0][E+][]

[s6,s3,s0][E+1][]

[s2,s6,s3,s0][E+1$][]

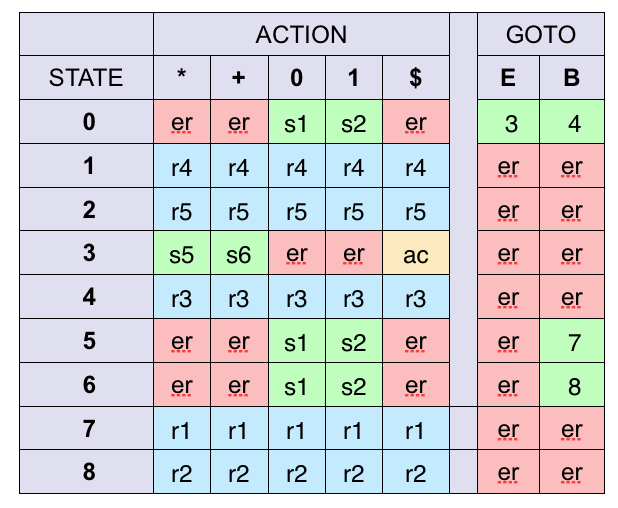
[s6,s3,s0][E+B$][5]

[s8,s6,s3,s0][E+B$][]

[s0][E$][2]

[s3,s0][E$][]

[s3,s0][E$][]



Given: 1+0\*1+0

[state][stack][transition]

// Checkthis

[s0][1][] // initial read, table[0][1] = push s2 onto stack

[s2,s0][1][5] // table[2][1] = Rule 5, replace 1 with B, pop s2

[s0][B][5] // table[0][B] = push s4

[s4,s0][B][5] // table[4][B] = Rule 3, replace B with E, pop s4

[s0][E][] // table[0][E] = push s3

[s3,s0][E][] // table[3][e] = er, read next symbol\*

[s3,s0][E+][] // table[3][+] = push s6

[s6,s3,s0][E+][] // table[6][+] = er, read next symbol\*

[s6,s3,s0][E+0][] // table[6][0] = push s1

[s1,s6,s3,s0][E+0][4] // table[1][0] = Rule 4, replace 0 with B, pop s1

[s6,s3,s0][E+B][] // table[6][B] = push s8

[s8,s3,s0][E+B][] // table[8][B] = er, read next symbol\*

[s8,s3,s0][E+B\*][] // table[8][\*] = Rule 2, replace E+B with E, pop s8

[s3,s0][E\*] // table[3][\*] = push s5

[s5,s3,s0][E\*] // table[5][\*] = er, read next symbol\*

[s5,s3,s0][E\*1] // table[5][1] = push s2

[s2,s5,s3,s0][E\*1] // table[2][1] = Rule 5, replace 1 with B, pop s2

[s5,s3,s0][E\*B] // table[5][B] = push s7

[s7,s5,s3,s0][E\*B] // table[7][B] = er, read next symbol\*

[s7,s5,s3,s0][E\*B+] // table[7][+] = Rule 1, replace E\*B with E, pop s7

[s5,s3,s0][E+] // table[5][+] = er, read next symbol\*

[s5,s3,s0][E+0] // table[5][0] = push s1

[s1,s5,s3,s0][E+0] // table[1][0] = Rule 4, replace 0 with B, pop s1

[s5,s3,s0][E+B] // table[5][B] = push s7

[s7,s5,s3,s0][E+B] // table[7][B] = er, read next symbol\* ... $ (beginning of input)

[s7,s5,s3,s0][E+B$] // table[7][$] = use R1... ???

\* LR(k) grammars can read k tokens ahead. Since this is an LR(1) grammar, we can read up to 1 more token.

~~[s0][1][] // initial read, table[0][1] = push state 2 onto stack~~

~~[s2, s0][1][] // table[2][1] = Rule 5, replace 1 with B, pop s2~~

~~[s0][B][5] // table[0][B] = goto state 4, Rule 3, replace B with E~~

~~[s0][E][3] // table[0][E] = goto state 3, error, read next symbol~~

~~[s0][E+][] // table[0][+] = error, read next symbol~~

~~[s0][E+0][] // table[0][0] = push state 1 onto stack~~

~~[s1, s0][E+0][0] // table[1][0] = Rule 4, replace 0 with B, pop s1~~

~~[s0][E+B][4] // table[0][B] = goto state 4, error, read next symbol~~

~~[s0][E+B\*][] // table[0][\*] = error, read next symbol~~

~~[s0][E+B\*1][] // table[0][1] = push state 2 onto stack~~

~~[s2, s0][E+B\*1][] // table[2][1] = Rule 5, replace 1 with B, pop s2~~

~~[s0][E+B\*B][5] // table[0][B] = goto state 4, error, read next symbol~~

~~[s0][E+B\*B+][] // table[0][+] = error, read next symbol~~

~~[s0][E+B\*B+0][] // table[0][0] = error, read next symbol~~

~~[s0][E+B\*B+0$][] // table[0][$] = error~~

### **Question 3**

Considering the following grammar for a simple programming language:

<program> ::= <statement> | <program> <statement>

<statement> ::= <assignStmt> | <repeatStmt> | <printStmt>

<assignStmt> ::= <id> = <expr> ;

<repeatStmt> ::= repeat ( <expr> ) <stmt>

<printStmt> ::= print <expr> ;

<expr> ::= <term> | <expr> <addOp> <term>

<term> ::= <factor> | <term> <multOp> <factor>

<factor> ::= <id> | <number> | - <factor> | ( <expr> )

show the **top-down** derivation of the following program:

firstvar = 1;

secondvar = 2;

repeat (10)

thirdvar = 2 \* (firstvar + secondvar) / (firstvar + 2);

repeat (firstvar + 2 \* secondvar)

repeat (thirdvar)

print firstvar;

**NOTE: Do not draw a tree! Show a linear sequence of transitions instead.**

<program>

->

<program> <statement>

<program> <program> <statement>

<statement> <program> <statement>

<assignStmt> <program> <statement>

<id> = <expr> ; <program> <statement>

<id> = <expr> ; <program> <statement>

id(firstVar) = <expr> ; <program> <statement>

id(firstVar) = <term> ; <program> <statement>

id(firstVar) = <factor> ; <program> <statement>

id(firstVar) = 1; <program> <statement>

id(firstVar) = 1; <program> <statement> <statement>

id(firstVar) = 1; <program> <statement> <statement>

id(firstVar) = 1; <program> <statement> <statement>

id(firstVar) = 1; <statement> <statement> <statement>

id(firstVar) = 1; <assignStmt> <statement> <statement>

id(firstVar) = 1; <id> = <expr> ; <statement> <statement>

id(firstVar) = 1; id(secondVar) = <expr> ; <statement> <statement>

id(firstVar) = 1; id(secondVar) = <factor> ; <statement> <statement>

id(firstVar) = 1; id(secondVar) = 2 ; <statement> <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat ( <expr> ) <stmt> <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat ( <term> ) <stmt> <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat ( <factor> ) <stmt> <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) <stmt> <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) <assignStmt> <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) <id> = <expr> ; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = <expr> ; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = <term> ; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = <term> <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = <term> <multOp> <factor> <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = <factor> <multOp> <factor> <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = <id> <multOp> <factor> <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 <multOp> <factor> <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* <factor> <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* (<expr>) <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* (<expr> <addOp> <term>) <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* (<term> <addOp> <term>) <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* (<factor> <addOp> <term>) <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( <id> <addOp> <term>) <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar <addOp> <term>) <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + <term>) <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + <factor>) <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + <id>) <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) <multOp> <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / <factor>; <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<expr>); <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<expr> <addOp> <term>); <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<term> <addOp> <term>); <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<factor> <addOp> <term>); <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<id> <addOp> <term>); <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> <addOp> <term>); <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> <addOp> <factor>); <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> <addOp> <id>); <statement>

id(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); <statement>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); <repeatStmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat ( <expr> ) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (<expr> <addOp> <term>) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (<term> <addOp> <term>) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (<factor> <addOp> <term>) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + <term>) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + <term> <multOp> <factor>) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + <id> <multOp> <factor>) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* <factor>) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* <id>) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* secondVar) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* secondVar) <repeatStmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* secondVar) repeat ( <expr> ) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* secondVar) repeat ( <term> ) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* secondVar) repeat ( <factor>) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* secondVar) repeat ( <id> ) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* secondVar) repeat (thirdVar) <stmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* secondVar) repeat (thirdVar) <printStmt>

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* secondVar) repeat (thirdVar) print <expr> ;

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* secondVar) repeat (thirdVar) print <term> ;

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* secondVar) repeat (thirdVar) print <factor> ;

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* secondVar) repeat (thirdVar) print <id> ;

àid(firstVar) = 1; id(secondVar) = 2 ; repeat (10) id(thirdVar) = 2 \* ( firstVar + secondVar) / (<firstVar> + 2); repeat (firstVar + 2 \* secondVar) repeat (thirdVar) print firstVar ;

### **Question 4**

Considering the following grammar for a simple programming language:

<program> ::= <statement> | <program> <statement>

<statement> ::= <assignStmt> | <repeatStmt> | <printStmt>

<assignStmt> ::= <id> = <expr> ;

<repeatStmt> ::= repeat ( <expr> ) <stmt>

<printStmt> ::= print <expr> ;

<expr> ::= <term> | <expr> <addOp> <term>

<term> ::= <factor> | <term> <multOp> <factor>

<factor> ::= <id> | <number> | - <factor> | ( <expr> )

show the **bottom-up** derivation of the following program:

firstvar = 1;

secondvar = 2;

repeat (10)

thirdvar = 2 \* (firstvar + secondvar) / (firstvar + 2);

repeat (firstvar + 2 \* secondvar)

repeat (thirdvar)

print firstvar;

**NOTE: Do not draw a tree! Show a linear sequence of transitions instead.**

**firstvar (term)**

**first var = (term) (assignStmt)**

**firstvar = 1 (term) (assignStmt) (id)**

**firstvar = 1 ; (term) (assignStmt) (id) ;**

**;**

**secondvar (term)**

**secondvar = (term) (assignStmt)**

**secondvar = 1 (term) (assignStmt) (id)**

**secondvar = 1 ; (term) (assignStmt) (id) ;**

**;**

**repeat (repeatStmt)**

**repeat id (repeatStmt)(id)**

**repeat id (repeatStmt)(id) ;**

**thirdvar (term)**

**thirdvar = (term) (assignStm)**

**thirdvar = 2 (term) (assignStm) (id)**

**thirdvar = 2 \* (term) (assignStm) (id) (multiOp)**

**thirdvar = 2 \* (firstvar (term) (assignStm) (id) (multiOp) (term)**

**thirdvar = 2 \* (firstvar + (term) (assignStm) (id) (multiOp) (term) (addOp)**

**thirdvar = 2 \* (firstvar + secondvar (term) (assignStm) (id) (multiOp) (term) (addOp) (term)**

**thirdvar = 2 \* (firstvar + secondvar) / (term) (assignStm) (id) (multiOp) (term) (addOp) (term) (factor)**

**thirdvar = 2 \* (firstvar + secondvar) / (firstvar (term) (assignStm) (id) (multiOp) (term) (addOp) (term) (factor) (term)**

**.............**

### **Question 5**

Modify the following grammar, so it is suitable for a recursive decent parser.

S → Sa | Sb | Sc | d | e | f

S → d | e | f | S’

S’ → Sa | Sb | Sc

### **Question 6**

Explain why a right-recursive grammar is not appropriate for bottom up LR(k) parsing.

Illustrate your answer with an example.

**{TODO}**

### **Question 7**

Explain why left-recursive grammar can be used neither for top-down LL(k) nor recursive decent parsing.

Illustrate your answer with an example.

**{TODO}**

SET 7

Names, Scopes, and Bindings

**Question 1**

Explain with details what a binding and a binding scope are.

What is the lifetime of a binding and what are the critical events in that lifetime?

Binding is the association of a name with something. A binding scope is the part of the program where the binding is active, whether it’s in a function or globally.

The lifetime of a binding is the period of time between the creation of the binding and its destruction. The critical events in its lifetime are the creation of objects and bindings, references to variables (which use bindings), the reactivation of bindings, and the destruction of objects and bindings.

**Question 2**

Explain with details what a binding time is.

What is the difference between static binding and dynamic binding.

What is the difference between early binding and late binding?

A binding time is the point at which a binding is created. The difference between static and dynamic binding is that static binding is based on the physical (lexical) structure of a program while dynamic binding is based on the current state of program execution. In other words, while static binding resolves to the most recent compile-time binding, dynamic binding requires that we choose the most recent binding at runtime.

In early binding, the referencing environment is bound when the reference is created (default for languages with static scoping). In late binding, the referencing environment is bound when the subroutine is called (default for languages with dynamic scoping).

**Question 3**

What are the main storage allocation mechanisms in the context of compilers?

Which parts of the program get statically allocated memory?

What the dynamically allocated memory is used for?

The main storage allocation mechanisms in the context of compilers are static, stack, and heap. Code, globals, static variables, explicit constants, and data structures with support information are all allocated statically. The stack is used for function calls and their associated variables. Dynamically allocated memory is used when a program requests a chunk of memory from the system, such as for the creation of a node in a linked list.

**Question 4**

Describe with details how procedures (a.k.a. functions, methods, etc.) are executed using the program stack.

Explain the role of the calling sequence, prolog, and epilog in stack management.

\

When a method is called,

**Question 5**

Explain with details what is the difference between internal fragmentation and external fragmentation in heap management?

Internal fragmentation refers to the unused parts of fixed-size blocks while external fragmentation refers to having enough free space for a request spread out across many individual chunks but none of the chunks being large enough to accommodate the allocation request individually.

**Question 6**

Consider the following requests (Rn) for memory allocation (A) and deallocation (D) from a heap of size 20:

R1 A 6

R2 A 3

R3 A 5

R4 A 2

R1 D

R5 A 4

R4 D

R6 A 1

R7 A 2

Compute explicitly (i.e., show all steps) the external fragmentation assuming the **best-fit** allocation policy.

best fit - find the smallest block smaller than n

Use the following notation:

[xxxxxxxxxxxxxxxxxxxx]

[111111xxxxxxxxxxxxxx]

[xxxxxxxxxxxxxxxxxxxx]

R1 A 6

[111111xxxxxxxxxxxxxx]

R2 A 3

[111111222xxxxxxxxxxx]

R3 A 5

[11111122233333xxxxxx]

R4 A 2

[1111112223333344xxxx]

R1 D

[xxxxxx2223333344xxxx]

R5 A 4

[xxxxxx22233333445555]

R4 D

[xxxxxx22233333xx5555]

R6 A 1

[xxxxxx22233333x65555]

R7 A 2

[xxxx7722233333x65555]

**Question 7**

Consider the following requests (Rn) for memory allocation (A) and deallocation (D) from a heap of size 20:

R1 A 6

R2 A 3

R3 A 5

R4 A 2

R1 D

R5 A 4

R4 D

R6 A 1

R7 A 2

Compute explicitly (i.e., show all steps) the external fragmentation assuming the **first-fit** allocation policy.

// first slot that can accommodate the request

Use the following notation:

[xxxxxxxxxxxxxxxxxxxx]

[111111xxxxxxxxxxxxxx]

[xxxxxxxxxxxxxxxxxxxx]

R1 A 6

[111111xxxxxxxxxxxxxx]

R2 A 3

[111111222xxxxxxxxxxx]

R3 A 5

[11111122255555xxxxxx]

R4 A 2

[1111112225555544xxxx]

R1 D

[xxxxxx2225555544xxxx]

R5 A 4

[5555xx2225555544xxxx]

R4 D

[5555xx33355555xxxxxx]

R6 A 1

[55556x33355555xxxxxx]

R7 A 2

[55556x3335555577xxxx]

**Question 8**

Consider the following requests (Rn) for memory allocation (A) and deallocation (D) from a heap of size 20:

R1 A 6

R2 A 3

R3 A 5

R4 A 2

R1 D

R5 A 4

R4 D

R6 A 1

R7 A 2

Compute explicitly (i.e., show all steps) the external fragmentation assuming the **worst-fit** allocation policy.

// use block with the most space

Use the following notation:

[xxxxxxxxxxxxxxxxxxxx]

[111111xxxxxxxxxxxxxx]

[xxxxxxxxxxxxxxxxxxxx]

R1 A 6

[111111xxxxxxxxxxxxxx]

R2 A 3

[111111222xxxxxxxxxxx]

R3 A 5

[11111122233333xxxxxx]

R4 A 2

[1111112223333344xxxx]

R1 D

[xxxxxx2223333344xxxx]

R5 A 4

[5555xx2223333344xxxx]

R4 D

[5555xx22233333xxxxxx]

R6 A 1

[5555xx222333336xxxxx]

R7 A 2

[5555xx22233333677xxxx]

**Question 9**

Consider the following code:

program scopes (input, output);

var x : integer;

procedure set(n : integer);

begin

x := n;

end;

procedure print;

begin

write(x);

end;

procedure first;

begin

set(1);

print;

end;

procedure second;

var x : integer;

begin

set(2);

print;

end;

 begin

set(0);

first;

print;

second;

print;

 end.

Explain the program behavior (i.e., how will it execute) and what it will print assuming static scope rules

Set 0 is called, setting global var x = 0. First is called, setting global var x = 1, and printing 1. Print is called again in main and 1 is printed. Then second is called and a new var x is declared. Set 2 is called and sets the global var x to 2. Print is called and prints 2. Then in main, print is called again and prints 2.

**1, 1, 2, 2**

**Question 10**

Consider the following code:

program scopes (input, output);

var x : integer;

procedure set(n : integer);

begin

x := n;

end;

procedure print;

begin

write(x);

end;

procedure first;

begin

set(1);

print;

end;

procedure second;

var x : integer;

begin

set(2);

print;

end;

 begin

set(0);

first;

print;

second;

print;

 end.

Explain the program behavior (i.e., how will it execute) and what it will print assuming dynamic scope rules

Set 0 is called, and sets the most recent binding of x (global var) to 0. Then first is called and set 1 is called, setting the most recent binding of x (global var) to 1. Print is called from within first and prints x (1).

Back in main, second is called and a new binding of x is created, after which set 2 is called. Set 2 sets the most recent binding of x (new variable from “second”) to 2. Then print is called and the last binding of x is called, so 2 is printed. Back in main, print is called, and 1 is printed. 1, 1, 2, 1

**Question 11**

Explain what a first-class, second-class, and a third-class objects are.

First-class objects are created at run-time, can be passed as parameters and returned from subroutines, can be assigned to variables and data structures, and have intrinsic identity independent from any name. Second-class objects cannot be created at runtime but references can be used and passed. Third-class objects may not be passed as a parameter, much less returned from a subroutine or assigned to a variable.

**Question 12**

Explain what a closure is and how it can be implemented.

A closure is the combination of a reference to a subroutine and an explicit representation of its referencing environment. For example, consider a function func1() with a Boolean variable “temp” inside, and a nested func2() inside of func1() which toggles the value of temp.

**Question 13**

Consider the following code:

program scopes (input, output);

var

x : integer := 3

y : integer := 2;

procedure multiply;

begin

x := x \* y;

end;

procedure second (procedure P);

var x : integer := 2;

begin

P;

end;

procedure first;

var y : integer := 3;

begin

second(multiply);

end;

 begin

first;

write(x);

 end.

Explain the program behavior (i.e., how will it execute) and what it will print assuming static scope rules.

In main, first is called and a new y var is created with value 3, then second is called. The multiply procedure is passed which sets x = x \* y. But the current scope has x = 2 and y = 3, so x (global) is set to 6.

Then once we are back at main, write(x) is called and the global var x (6) is printed.

**Question 14**

Consider the following code:

program scopes (input, output);

var

x : integer := 3

y : integer := 2;

procedure multiply;

begin

x := x \* y;

end;

procedure second (procedure P);

var x : integer := 2;

begin

P;

end;

procedure first;

var y : integer := 3;

begin

second(multiply);

end;

 begin

first;

write(x);

 end.

Explain the program behavior (i.e., how will it execute) and what it will print assuming dynamic scope rules with shallow binding.

In shallow (late) binding, the referencing environment is bound when the subroutine is called

First is called and a new var y is set to 3. Then second is called a the multiply procedure is passed, which sets x = x (2) \* y(3). But x (last active binding) was from the second procedure, so the global var x is still 3, so back in main, 3 prints.

**Question 15**

Consider the following code:

program scopes (input, output);

var

x : integer := 3

y : integer := 2;

procedure multiply;

begin

x := x \* y;

end;

procedure second (procedure P);

var x : integer := 2;

begin

P;

end;

procedure first;

var y : integer := 3;

begin

second(multiply);

end;

 begin

first;

write(x);

 end.

Explain the program behavior (i.e., how will it execute) and what it will print assuming dynamic scope rules with deep binding.

// In deep (early) binding, the referencing environment is bound when the reference is created

**Question 16**

Explain the following terms:

overloading

coersion

casting

polymorphism

explicit parametric polymorphism

implicit parametric polymorphism

sub-type polymorphism

Overloading refers to using one name to refer to numerous entities. Coercion, or autocasting, is the process in which the compiler automatically converts a value of one type into another type if needed by the surrounding context. Casting is the explicit instruction to convert a type. Polymorphism refers a value having many types and it comes in two forms: parametric and subtype. Explicit parametric is known as genericity (generics) and refers to code that takes a type (or a set of types) as a parameter. Sub-type polymorphism refers to inheritance and code that is designed to work with values of some type T, where the programmer can define additional types to be extensions/refinements of T, and the polymorphic code will also work with these subtypes. Implicit parametric polymorphism may be implemented either way.

**Question 17**

Explain with details what is a symbol table and how it can be implemented for both static and dynamic scoping rules.

In static scoping rules, a symbol table is a data abstraction that the compiler uses to map names to the information that the compiler knows about them. The symbol table is used to track bindings at compile time.

However, in dynamic scoping, the bindings must be tracked at run-time using an association list or a central reference table. Association lists (A-lists) are simple but can be inefficient, while central reference tables require more work to update, but they are efficient for lookup operations.

SET 8

Semantic Analysis

**Question 1**

Describe the role of semantic analysis in program compilation.

What does a semantic analyzer do?

The main role of a semantic analyzer is to enforce static semantic rules. It participates in the construction of an abstract syntax tree, gathers information needed by the code generator, performs dynamic checks (such as out-of-bounds, memory violation, pointer/reference issues, and others), and verifies assertions (such as invariants, preconditions, and post-conditions).

**Question 2**

What is a Cambridge Polish Notation?

Write the following expression using s-expressions:

f(x, y + z, g(s, t)) \* h(a, c + d \* e / p((a \* b / c), r(s, t))) + m \* n \* k \* f(a, b, c)

Cambridge Polish Notation is a notation where both the function and the operand(s) are enclosed in parentheses.

add( mult (f(x, sum(y, z), g(s, t)), h(a, sum(c, mult(d, div(e, p(div (mult(a, b), c))))), r(s, t))), mult(mult(mult(m, n), k), f(a, b, c)))

**Question 3**

What is an attribute grammar? Describe what grammar attributes are used for.

What types of attributes do you know? Describe the use of each of the types.

Define an S-attributed grammar. Can an LL(k) grammar be S-attributed?

Attribute grammars serve to define the semantics of a program by adding attributes to symbols. The attributes are not assignments; they are definitions that define the left-hand side of the rule in terms of the right-hand side of the rule.

**Question 4**

Design an S-attributed tree grammar that can be used to build syntax trees for programs that are defined by the following LR(1) grammar:

(1) E → E \* B

(2) E → E + B

(3) E → B

(4) B → 0

(5) B → 1

Show how your MODIFIED grammar builds a syntax tree for the following program:

1+0\*1

using the following notation:

[production] (<current tree using s-expressions>)

production is just one of the numbers 1 to 5 pointing to the production used in the derivation, and as an example of a tree represented by an s-expression, consider that this tree

+

/ \

\* \*

/ \ / \

1 0 0 1

would be represented by the following s-expression:

(+ (\* (1) (0)) (\* (0) (1)))

Please show ALL productions from the derivation; if there are no changes to the tree, then you may skip the second part.

(1) E1.val → E2.val \* B.val {E1.val = E2.val \* B.val}

(2) E1.val → E2.val + B.val {E1.val = E2.val + B.val}

(3) E → B {E.val = B.val}

(4) B.val → 0 {B.val = const.val}

(5) B.val → 1 {B.val = const.val}

**Question 5**

Define an L-attributed grammar. What is it used for? What is a "decoration"? What are "semantic rules"?

Can an LR(k) grammar be L-attributed?

Can an LL(k) grammar be L-attributed?

**{TODO}**

**Question 6**

Design an L-attributed tree grammar that can be used to build syntax trees for programs that are defined by the following LL(1) grammar:

(1) S → F FT

(2) FT → + F FT

(3) FT → <empty>

(4) F → ( S )

(5) F → 1

Show how your grammar builds a syntax tree for the following program:

((1+1)+1)

using the following notation:

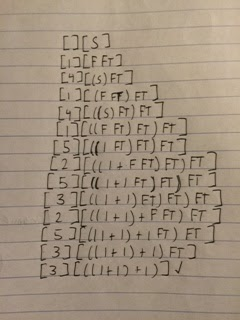
[production] (<current tree using s-expressions>)

For example, ((1+1)+1) would be represented as:

(+ (+ (1) (1)) (1))

NOTE: show ALL productions from the derivation.

//HOLLABACKGURL



What are action routines?

Describe the use of action routines in compilation of programs. How does it differ from using attribute grammars?

**{TODO}**

SET 9

Flow Control

**Question 1**

Explain the role of side effects in imperative languages.

Side effects are important in imperative languages because they can change the environment where they operate in without having to return a value, influencing subsequent computation.

A programming language construct is said to have a side effect if it influences subsequent computation and ultimately program output in any way other than by returning a value for use in the surrounding context. Assignment to variables is perhaps the most fundamental side effect. Imperative languages have statements which are executed solely for their side effects and return no useful value.

**Question 2**

Analyze the following code:

matrix[getRow(), getColumn()] := matrix[getRow(), getColumn()] + 1;

If we insist that getRow() and getColumn() are idempotent functions, will the code be correct?

Write a version of this snippet that will work even if the functions are not idempotent.

// *idempotent* : if called repeatedly with the same set of arguments, it will always return the same value, and the number of consecutive calls (after the first) will not affect the results of subsequent execution.

The code may be correct if the functions getRow() and getColumn() have no side-effects, but this isn’t good practice. New version:

Matrix[getRow(), getColumn()] += 1;

**Question 3**

Explain what is the difference between value-oriented languages and reference-oriented languages.

Use a code example and memory map to explain the difference.

Value-oriented languages consider variables to be containers for data while reference-oriented languages consider variables to be containers for references to containers that hold data.

For example, Java’s built-in types are value oriented.

int a = 2;

[a] -> 2

However, objects are reference-oriented.

Object obj = new Object();

[obj] -> [0xabcd] -> [obj data]

**Question 4**

Explain the difference between expression-oriented and statement-oriented languages.

Is the following expression-oriented or statement-oriented (justify your answer):

1 C, 2 Java, 3 ciLisp v.1

Expression-oriented languages return values from expressions while statement-oriented languages generally may also rely on the side-effects produces by a statement. Most imperative languages are statement-oriented.

C and Java are definitely statement-oriented because they can have methods whose sole purpose is to produce side-effects which we can use to compute later (such as rand()).

ciLisp v.1 is expression-oriented because it produces return values and doesn’t rely on side-effects to produce results.

**Question 5**

Compute the value of the following expression assuming that all used operators are right-associative.

18-4+2-9+1-7-5

18-4+2-9(-7-5)

18-4+2(-9-13)

18-4(+2-22)

18(-4-20)

18-24

-6

**Question 6**

Compute the value of the following expression assuming that all used operators are left-associative.

18-4+2-9+1-7-5

(18-4)+2-9-7-5

(14+2)-9-7-5

(16-9)-7-5

(7-7)-5

-5

**Question 7**

Explain what short-curcuiting of boolean expressions is.

Assuming that the language supports short-circuit evaluation of boolean expressions, and that expressions are evaluated from left to right, which one of the following snippets of code is most likely more efficient.

1 if ((a != 0) && f1() && f2()) ...

2 if (f1() && (a != 0) && f2()) ...

3 if (f1() && f2() && (a != 0)) ...

Justify your answer.

Short-circuiting of a Boolean expression is the process of terminating a compound condition statement (a condition statement with more than 1 condition) if a single condition is false. We can think of this as a chain where each Boolean statement is a link. If a single statement is false, the chain is breaks and the whole expression evaluates to false.

The 1st snippet is the most efficient because it only has to compare a with 0. If a == 0, then the if-condition can terminate immediately due to short-circuiting. Snippets 2 and 3 call f1() which is more computationally expensive than comparing a with 0.

**Question 8**

Explain how

1 true iterators,

2 iterator objects, and

3 1st-class functions

are used for iteration.

A true iterator is a separate thread of control, with its own program counter, that decouples algorithm used for the enumerating of elements from the code that will use the elements.

An iterator object can also iterate although between calls, but the state of the iterator must be kept in the object’s data members.

Iterating with first-class functions can be done by writing the body of the loop as a function with a loop index as an argument, and then passing the function to an iterator.

**Question 9**

When do we say that a statement "threatens" an index variable?

Explain why including such statements in loops constitutes a problem.

Write some C code to illustrate the problem.

A statement threatens an index variable if there is a chance that it may assign to it, read it from a file, pass it to another function, or do any action that may cause the index variable to change in any other way than just incrementing/decrementing to reach the next element.

Example: Remove even numbers from an array

main {

int n = 4;

int array[4] = {3, 6, 9, 12};

int i = 0;

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

if(array[i] % 2 == 0) {

array[i] = 1;

i = 0;

}

}

}

**Question 10**

Explain why in many cases using a switch (a.k.a. case) statement is better than using the if...then... construct.

Often, a switch statement is better because the compiler can produce a jump table to the appropriate memory address in the case that a condition is met.

**Question 11**

What might be a problem when using a recursion? In this context, explain what a tail recursion is and why it is better than embedded version.

A potential problem with recursion is the reliance on the stack to keep track of function calls and even local variables, possibly resulting in a stack overflow. This can be alleviated with a tail-recursive function; a compiler can recognize a tail recursive function, allowing it to assign a return address to somewhere within the function (similar to GOTO), thereby “recycling” stack space instead of creating new stack frames.

**Question 12**

Write a tail-recursive version of the following function:

char \*array[] = {"one", "two", "three", "four", "five", NULL};

int count\_iter(char \*\*elem) {

int count = 0;

while (\*elem) {

count++;

elem += 1;

}

return count;

}

int count\_TR\_tail\_helper(char \*\*elem) {

// check if array has 1 element

if(\*elem)

count\_iter\_tail(elem + 1, 1);

else

return 0;

}

int count\_TR\_tail(char \*\*elem, int count) {

if(\*elem == NULL)

return count;

else return count\_iter\_tail(elem + 1, count + 1);

}

**Question 13**

Write an embedded-recursive version of the following function:

char \*array[] = {"one", "two", "three", "four", "five", NULL};

int count\_iter(char \*\*elem) {

int count = 0;

while (\*elem) {

count++;

elem += 1;

}

return count;

}

int count\_ER(char \*\*elem) {

if(\*elem == NULL)

return 0;

else return 1 + count\_ER(elem + 1);

}

SET 10

Data Types

**Question 1**

Define the following terms and give examples of languages in each category:

1. strongly-typed language

2. statically-typed language

3. dynamically-typed language

A strongly-typed language is implemented in such a way that it doesn’t allow certain operations on an object if it isn’t intended to support it. Thus, there are no loopholes around the type system. Some strongly-typed languages are Ada and C.

Statically-typed languages are also strongly-typed, although object types are checked at compile time (implying that all variables must have a type declared within the source code). Some statically-typed languages are Ada and Pascal.

Dynamically-typed languages wait to check variable types until run-time. Examples are Perl and Ruby.

### **Question 2**

Consider the following type declarations:

struct one {int k; float f;}

struct two {int size; float average;}

Are these two types equivalent?

Justify your answer thoroughly.

The two types are structurally equivalent since they have the same fields, however they are not name equivalent so they are technically two different types. Thus, when considering name equivalence, the two types are not the same.

### **Question 3**

Consider the following type definitions:

typedef MY\_FLOAT float;

typedef YOUR\_FLOAT float;

MY\_FLOAT f1 = 0.5;

YOUR\_FLOAT f2 = f1;

Should this code compile correctly? Disregard that this is C, and instead provide a generalized discussion using proper terms as used in the study notes.

If a language has strict name equivalence, where each unique alias is a unique type, then the code won’t compile correctly. However, if the language has loose name equivalence, then the code should compile because even though the names are different, the structure of the types is the same.

### 

### **Question 4**

Compute the memory requirements for each of the two following declaration of arrays:

char months\_1[][10] = { “January”, “February”, “March”, “April”, "May”, “June”, “July” , "August", "September", "October", November", "December"};

char \*months\_2[] = { “January”, “February”, “March”, “April”, "May”, “June”, “July” , "August", "September", "October", November", "December"};

Explain how did you obtain the values.

char months\_1 is contiguously allocated, thus the array is “square” and the length of all rows is the same length as the longest string. Thus, the size of the array is (10 \* 12) bytes. ((each character in September + a null terminator) \* 12 rows). This results in a total of 120 bytes.

char \*months\_2 is an array of pointers to a single array wherein all of the month names are stored, however the array is “ragged”. In a 32-bit system, each pointer is 4 bytes. Storage for this array is the following:

36 (12 months \* 4 bytes per pointer)

+ 72 (character bytes in all the months)

+ 12 (null terminators)

= 120 bytes

**Question 5**

Consider the following declaration:

struct test {

int id;

char char\_resp[14];

union {

int int\_resp;

double dbl\_resp;

}

short short\_resp;

}

Describe possible internal representations of the struct assuming a 32-bit system architecture:

1. naive,

2. optimized for space,

3. optimized for memory alignment.

How much space does each representation need?

Assuming the 32-bit system is 4-byte aligned, X is an empty byte. Struct members are allocated in the order that they appear.

1) Naive allocation

[ id ]

[ char\_resp[0-3] ]

[ char\_resp[4-7] ]

[ char\_resp[8-11] ]

[ char\_resp[12-14]X ]

[ dbl\_resp(1st half)] // For union, allocate space for the

[ dbl\_resp(2nd half)] // largest member (double => 8 bytes)

[ [short\_resp]XX ]

2) Allocation optimized for space

[\*\*\*\*\*id\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*]

[\*\*\*\*\*char\_resp[0-3]\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*]

[\*\*\*\*\*char\_resp[4-7]\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*]

[\*\*\*\*\*char\_resp[8-11]\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*]

[\*\*\*\*\*char\_resp[12-14]|dbl\_resp(0th byte)\*\*\*\*\*\*\*\*]

[\*\*\*\*\*dbl\_resp(bytes 1-4)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*]

[\*\*\*\*\*dbl\_resp(bytes 5-7)|short\_resp(0th byte)\*\*\*]

[\*\*\*\*\*short\_resp(1st byte)XXX\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*]

3) Allocation optimized for memory alignment (minimize holes)

[ id ]

[ char\_resp[0-3] ]

[ char\_resp[4-7] ]

[ char\_resp[8-11] ]

[ char\_resp[12-14]|X ]

[ dbl\_resp(bytes 0-3) ]

[ dbl\_resp(bytes 4-7) ]

[ short\_resp|XX ]

### **Question 6**

Consider the following snippet of C code:

int i = 0, \*p = 0;

print("i = %d", i + 1);

print("p = %d", p + 1);

Tell what will be printed without running the code. Justify your answer.

The first print statement will print 1, since int i is assigned 0. Int pointer p is also assigned 0, but due to pointer arithmetic, adding 1 will increment the pointer’s value by 4, therefore the second print statement will print 4.

### **Question 7**

Describe the mark-and-sweep garbage collection algorithm. Explain the role of type descriptors in the algorithm.

The mark and sweep algorithm has 3 steps. First, the collector traverses the heap and marks every block as useless. Then, starting from outside of the heap, the collector recursively traverses through the linked data structures and marks each newly discovered block as useful. (Due to the nature of recursion, this second step may need require the program to allocate even more memory, although a pointer-reversal algorithm can be implemented to backtrack without having to allocate new memory.) Finally, the collector traverses the heap again and moves every block that is still marked as useless to the free list.

Type descriptors play an important role; by placing one near the beginning of each block, the collector knows where every “in-use” block begins and ends, and can also find the pointers contained within each block.

### **Question 8**

Describe how locks and keys are used to address issues with dangling pointers.

[answer]

**{TODO}**

### **Question 9**

Describe how tombstones are used to address issues with dangling pointers.

Tombstones add an extra level of indirection to pointers. Instead of having a pointer reference an object directly, it can reference a tombstone which contains the address of the object. When the object is freed, the tombstone can reset it’s field containing the address of the object to some pre-determined sentinel value so that future references to the now freed object will see the sentinel value instead.

### **Question 10**

Describe the use of reference counts in garbage collection. Explain the role of type descriptors in the algorithm.

**{TODO}**

### **Question 11**

Describe how the conservative garbage collection works. Why would one want to use such a scheme?

**{TODO}**

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### **Question 12**

Consider the following C code:

#include <stdlib.h>

#define SIZE\_OF\_DATA 32

#define NUM\_OF\_NODES 5

typedef struct list {

char \*data;

struct list \*next;

} LIST;

void allocateList(LIST \*\*list, int count) {

if (count == 0) {

\*list = NULL;

return;

}

\*list = (LIST\*)malloc(sizeof(LIST));

(\*list)->data = (char\*)malloc(SIZE\_OF\_DATA);

allocateList(&((\*list)->next), --count);

}

void freeList(LIST \*head) {

// TO BE IMPLEMENTED

}

int main(void) {

LIST \*head = NULL;

allocateList(&head, NUM\_OF\_NODES);

freeList(head);

}

Write a tail-recursive version of freeList() function.

Explain the potential for compiler optimization of such functions.

int freeList\_TR(LIST \*head) {

if(head == NULL)

return 0;

LIST \*to\_free = head->next;

free(head);

return freeList\_TR(to\_free);

}

The compiler can optimize the function by creating a go-to statement within the function so that instead of allocating new stack frames for a “normal” recursive function, the existing stack frame is repurposed for further computation.

SET 11

Subroutines and Control Abstractions

**Question 1**

Describe the stack layout supporting subroutine calls.

Then, describe with details the mechanism of calling subroutines (including descriptions of the prologue and epilogue and all stack manipulations).

The stack layout supporting subroutine calls consists of an activation record. It contains the arguments passed to the subroutine, return values, bookkeeping information, local variables, temporaries, as well as a frame pointer containing the address of the stack frame, and a return address to the caller.

Calling a subroutine has 2 phases: the prologue and the epilogue.

During the prologue, the caller passes arguments, saves the state of the registers and PC, and passes the return address and changes the PC. The callee allocates a stack frame, changing the stack and frame pointers while saving any values needed to return (e.g. return address).

During the epilogue, the callee passes return values, deallocates the stack frame (while restoring the stack and frame pointers), and restores the PC. The caller moves the return values where they are needed and restores the registers.

**Question 2**

Define formal and actual parameters. Describe what they are used for.

Write a sample short C code that illustrates the difference. Provide appropriate detailed commentary to make your point.

Do parameters necessarily need to match by position? What exemptions do you know?

A formal parameter is a parameter name in a subroutine declaration while an actual parameter is the actual value being passed to a subroutine.

In the example below, ‘val’ is a formal parameter while ‘a’ is an actual parameter.

#include <stdio.h>

void bar(int \*val){ (\*val) = (\*val) + 1; }

int main(){

int a = 1;

printf("a = %d\n", a); // prints 1

bar(&a);

printf("a = %d\n", a); // prints 2, a was modified

return 0;

}

Only some languages support parameters being in different positions (called keyword parameters), such as Ada and Python. However, many languages allow a variable number of parameters.

**Question 3**

Consider the following declarations of subroutines:

procedure A

procedure B

{...}

procedure C

procedure D {...}

procedure E {...}

{...}

procedure F

procedure G {...}

procedure H {...}

{...}

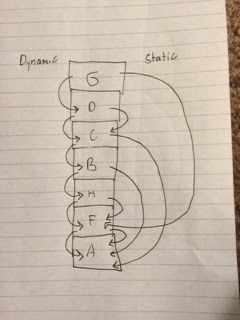
{...}

Show the stack (including dynamic and static links) after the following code is executed:

A; F; H; B; C; D; G;

Provide appropriate detailed commentary.

//CORDAWG5000



**Question 4**

Define precisely what passing parameters by value and passing parameters by reference are.

Write a short C program that illustrates the difference. Provide appropriate detailed commentary.

By passing a parameter by value, a function foo() passes a value to function bar() but bar() operates on a copy of foo()’s parameter. When bar() finishes, bar()’s original parameter is unchanged.

If foo() passes a parameter by reference to bar(), bar() has the memory address of the parameter so any changes that bar() makes to the parameter will still be there after bar() finishes because it is operating directly on the parameter (not a copy of it).

For example:

#include <stdio.h>

void foo(int val){

val += 1; // adds 1 to a copy of a

}

void bar(int \*val){

(\*val) = (\*val) + 1; // add 1 to parameter a itself

}

int main(){

int a = 1;

printf("a = %d\n", a); // prints 1

foo(a);

printf("a = %d\n", a); // prints 1, a wasn’t modified

bar(&a);

printf("a = %d\n", a); // prints 2, a was modified

return 0;

}

**Question 5**

Is the following macro safe? Support your answer with an example.

#define ABS(x) (((x) < 0) ? -(x) : (x))

Write an equivalent inline function that implements the same functionality. Is the inline function safe? Justify your answer.

The C preprocessor expands a macro while the compiler handles an inline function. The text expansion of macros can cause bugs. The macro is unsafe because there is no type checking and a parameter may be evaluated twice (in the case of ABS(x++)). In contrast, an inline function evaluates a parameter exactly once making it safe.

inline double ABS(double a) { return (a < 0) ? -a : a; }

**Question 6**

Explain what code in-lining is. How does it differ from pre-processor macros?

What is the main advantage?

What is the main disadvantage?

The C preprocessor expands a macro while the compiler handles an inline function. The text expansion of macros can cause bugs making inline functions semantically preferable. The main advantages of code inlining are that the overhead of calling a function can be avoided and we can be sure that the side effects of that function occur only once.

**Question 7**

Explain what a cactus stack is and what is it used for.

Explain how co-routines are different from threads.

A cactus stack is a method for allocating co-routines on a stack so that the program can transfer from one co-routine to another. Co-routines are different than threads in that they require an explicit “transfer” operation to go from one to another.

**Question 8**

Describe with details how exceptions can be implemented and how are they propagated.

How to improve efficiency of the exception mechanism?

Exceptions can be implemented with a linked stack of handlers; when a protected block is entered, the associated handler is added to the head of the list. However, if an exception is raised and a handler doesn’t match it, then it must re-raise the exception until the appropriate handler is found. This approach has run-time overhead since every time protected block needs to push a handler onto the handler list and then pop it back off.

Another implementation is to use a table with two fields; the starting address of a block of code and its associated handler. By sorting the table by the former, the handler for a code block can be found using the program counter as the key through a binary search. Although the cost of a lookup is great, assuming that exceptions rarely occur, this implementation is faster since the run-time overhead of a linked stack implementation is avoided.

**Question 9**

Describe what event-driven (or event-based) programming is.

Write a short C code that illustrates the concept. Provide appropriate detailed commentary.

[Pg 434 book (465 in pdf)]

Event-driven programming is a paradigm intended to allow a program to respond to events that occur outside of it, such as mouse clicks or keystrokes, which may occur at any time.