https://folk.ntnu.no/michaeng/tdt4205/
michael.engel@ntnu.no

Example Exam Questions for Compiler Construction
Spring 2020

Example Exam Questions

1 Compiler Structure (10 points)

The first phases of a compiler consist of lexical analysis (scanner), syntactic analysis (parser) and semantic analysis. For the given rules of a programming language, specify which phase of the compiler should verify that the program adheres to that rule and give a short explanation why this phase of the best one to perform this check.
If a rule could be checked equally well in different phases, discuss the tradeoffs briefly.

1.1. A function call has the correct number of arguments.

1.2. Digits `[0-9]` may appear in identifiers, but not as the first character (e.g., `a123` and `pi314` are valid identifiers, but `2big` is not).

1.3. Every variable must be declared before it is used in the program (as in C).

1.4. Assignments such as `a=42;` must end with a semicolon `(;)`.

2 Context-Free Languages (10 points)

Write context free grammars for the following languages (your grammar does not have to be LR(1), LL(1) etc):

2.1. All strings open and close parentheses, where the parentheses are balanced.

2.2. The language described by the regular expression `((ab)*(c|d))*`.

2.3. Expressions consisting of `num`, `+`, and `*`. Design your grammar so that `*` has higher precedence than `+`.

3 Formal Languages and Automata (10 points)

Consider the language L consisting of all strings w over the alphabet $\{\mathtt{q}, \mathtt{u}\}$ such that w contains qq or uuuu (i.e., at least 2 q's in sequence or at least 4 u's in sequence **or both**). For example, the strings uquqq, qqqquq, quuuuq, qqq, etc. belong to the language L.

3.1. Construct a regular expression for the language L.

3.2. Construct an NFA recognizing L from the regular expression.

3.3. Construct a DFA recognizing L either by deriving it from the NFA of question 3.2 or by constructing one directly.

3.4. Give an example of a formal language that is not context-free.

4 LR Parsing (15 points)

Consider the following grammar:

```
0: S → X
1: X → a X c
2: X → X X
3: X → b
```

4.1. What is Closure({X → X ↑ X})?

4.2. What is Goto({X → a ↑ X c}, X)?

4.3. Show the execution of the parser on the string abbc. The state machine for the parser is provided along with a table for you to fill in.

State	a	b	c	$	S	X
1	s2	s3			g5	g4
2	s2	s3				g6
3	r3	r3	r3	r3		
4	s2	s3		r0		g7
5				accept		
6		s3	s8			g7
7	r2	r2	r2	r2		
8	r1	r1	r1	r1		

Fill in the table to the right. In one line, show the current status of the parser -— the position in the input, the state the parser is in, and the contents of the stack. In the next line, show the action that the parser takes. Then show the new status in the following line. Repeat this process until the parser accepts the input. The first two are already filled in as an example.

Input	State	Stack
↑ a b b c	1	
		Shift to state 2
a ↑ b b c	2	a_1
		Shift to state 3
a b b c		
a b b c		
a b b c		
a b b c		
a b b c		
a b b c		
a b b c		
a b b c		
a b b c		
a b b c		
a b b c		
a b b c		
a b b c		

5 Parse trees and ASTs (10 points)

The following grammar is given:

```
 1 Start  → Expr
 2 Expr   → Expr + Term
 3        | Expr - Term
 4        | Term
 5 Term   → Term × Factor
 6        | Term ÷ Factor
 7        | Factor
 8 Factor → "(" Expr ")"
 9        | number
10        | ident
```

5.1. Draw the parse tree for the input string `2×a+b÷2×a`.

5.2. Reduce the parse tree to obtain the abstract syntax tree (AST).

5.3. Avoid code duplications in the AST by deriving a directed acyclic graph (DAG) from the AST.

6 Control Flow Graphs (10 points)

The following code is given:

```
x = 0;
i = 1;
while (x < 100) {
  if (a[x+i] > 0)
    i = x + a[i];
  x = x + 1;
}
print(i);
```

6.1. Draw the control flow graph (CFG) for this piece of code.

6.2. Draw the dependence graph for the code.

7 Procedures (10 points)

The following code is given:

```
int oho(int n) {
  return n * 2; // ←
}

int bar(int q, int r) {
  int s;
  s = r;
  if (s > q) {
    return bar(q, r-1);
  } else
    return oho(q-1);
  }
}

int foo(int a) {
  int x = 42;
  x = bar(a, a+2);
}
```

Hint: You can give dummy values for the return addresses on the stack in the following two questions.

7.1. Describe the structure of the activation record of function `foo`.

7.2. Give the **complete** contents of the stack when program execution arrives at the line indicated by the arrow (←) Give variable names and concrete values if known, otherwise give the name and a formula for the variable value (e.g. `q = a+b-4`).

8 Converting to x86 Assembler (10 points)

The following code is given:

```
int i, j;
i=23;
j=42;

while(i) {
  i = i-1;
  if (i<10) {
    j = j-1;
    foo(j, i);
  }
}
```

Hint: Compiler generated output is, of course, not accepted...
Convert the given code to x86-64 assembler code. The function `foo` is not given here and doesn't need to be implemented. Take care to use the correct registers for parameter passing.

9 yacc and lex (15 points)

9.1. Draw a diagram to visualize the interaction of a lex scanner and yacc parser (Example: `Text` → `function1()` → `function2()`).

9.2. A compiler can also be implemented without an explicit scanner. Then, the parser would read all input characters by itself (e.g., using `getchar`). Discuss the advantages and disadvantages of such an approach in three to four sentences.

9.3. The following lex code implements a skeleton scanner that ends parsing when the input `end` is read. Extend this code to create a scanner that detects positive integer numbers in the input and outputs the sum of the numbers. Use whitespace as separators between numbers and other text.

```
%{
#include <stdio.h>
enum { END = 256 };
%}
%%
end  { return (END); }
.    { printf("%c", yytext[0]); }
%%
int main(void) {
  int token;

  while (1) {
    token = yylex();

    if (token == END)
      break;
  }
  printf("\nBye!\n");
  return 0;
}
```