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

Exam for Compiler Construction
Eksamen kompilatortekknik

1 Regular Languages / Regulære språk (10 points / poeng)

1.1. Write a regular expression for all strings of `a`'s and `b`'s which contain the substring `abba`.

Skriv ned et/eit regulær uttrykk for alle streng til `a`'s og `b`'s som inneholder/inneheld delstringen `abba`.

1.2. Write a regular expression for all strings of `x`'s and `y`'s where every `y` is immediately followed by at least 3 `x`'s.

Skriv ned en/ein regulær uttrykk for alle streng til `x`'s og `y`'s der hvert/kvart `y` er umiddelbart fulgt av minst 3 `x`.

1.3. Write a regular expression for all strings of `p`'s and `q`'s which contain an odd number of `q`'s.

Skriv ned en/eit regulær uttrykk for alle streng til `p`'s og `q`'s som inneholder/inneheld et odde antall/talet på `q`.

1.4. A *finite langauge* is a language with a finite number of strings. For example, the language with only the strings `a`, `ba`, and `bba` is finite, while the langauges in question 1.1, 1.2, and 1.3 above are not finite. Are all finite languages regular? If so, explain why. If not, give an example of a finite langauge which is not regular.

Et/Eit *endelig/endeleg språk* er et/eit språk med et/ei endelig antall/mendge strenger. For eksempel/Til dømes er språket med bare/berre strengene `a`, `ba` og `bba` endelig, mens/medan ikke/ikkje de i spørsmålet 1.1, 1.2 og 1.3. ovenfor/ovanfor. Er alle endelige/endelege språk regulær? I så fall, forklar hvorfor/kvifor. Hvis/Viss ikke, gi et eksempel/ei døme på en endelig språk som ikke/ikkje er regulær.

2 NFAs and DFAs / NFA og DFA (10 points / poeng)

2.1. Convert the NFA given in fig. 1 to a DFA.

Konverter NFA vist i fig. 1 til en/ein DFA.

2.2. Give the corresponding regular expression.

Skriv ned tilsvarende/tilsvarande regulære uttrykk.

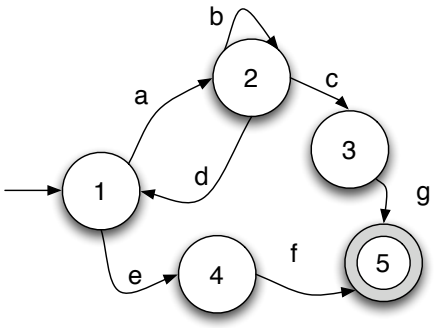

Figure 1: NFA

3 Compiler Structure / Kompilatorstruktur (10 points / poeng)

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

De/Dei første faser/fasene av en/ein kompilator består av leksikalisk analyse (scanner), syntaktisk analyse (parser) og semantisk analyse. For de/dei reglene/reglane for et/eit programmeringsspråk, spesifiser i hvilket/noko som fase kompilatoren skal sjekke/sjekka at programmet forholder/forheld seg til denne regelen, og gi en kort forklaring hvorfor/kvifor denne fasen er den beste for å utføre/utføra denne.
Hvis en/Viss ein regel kan sjekkes/sjakkast like godt i forskjellige/ulike faser, må du diskuterei/diskutera *tradeoffs* kort.

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

En funksionskall har riktig antall argumenter.

3.2. 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).

Sifrene/Siffera `[0-9]` kan vises/visast i identifikatorer, men ikke/ikkje som det første tegnet/teikna (f.eks. `a123` og `pi314` er gyldige identifikatorer, men `2big` er ikke/ikkje).

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

Hver variabel må deklareres/deklarerast før den brukes/han blir brukt i programmet (som i C).

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

Tilordne/Tilordn som `a=42;` må avsluttes/blir avslutta med et/eit semikolon `(;)`.

4 LL Parsing (10 points / poeng)

Consider the following grammar:
Se/Sja på følgende/følgjande grammatikken:

```
S → S a S b
  | c
  | Q q
Q → Q m
  | ε
```

4.1. Which non-terminals (if any) can derive the empty string?

Hvilke/Kva *non-terminaler* (hvis noen/viss nokon) kan utlede den tomme stringen?

4.2. What are the `FIRST` sets of `Q` and `S`?

Hva/Kva er de `FIRST` settene/setta av `Q` og `S`?

4.3. What are the `FOLLOW` sets of `Q` and `S`?

Hva/Kva er de `FOLLOW` settene/setta av `Q` og `S`?

4.4. This grammar can not be parsed by an LL(0) or LL(1) parser. Explain why not.

Denne grammatikken kan ikke/ikkje parses med en LL(0) eller LL(1) parser. Forklar hvorfor/kvifor ikke.

4.5. Rewrite the grammar so that it accepts the same language, but can be parsed by an LL(1) parser (use by left factoring and eliminate left recursion).

Omskriv grammatikken slik at den aksepterer det samme/same språket, men kan parses med en LL(1)-parser (bruk *left factoring* og *eliminate left recursion*).

5 Parse trees and ASTs / Parse-treer og ASTer (10 points / poeng)

The following grammar is given:
Se/Sja paå følgende/følgjande grammatikken:

```
 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 `b×c+3÷c×b`.

Tegn parsertreet/Teikn parsartreet for string `b×c+3÷c×b`.

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

Reduser parsertreet/parsartreet, og skriv ned abstrakte syntaks-treet (AST).

5.3. Derive a directed acyclic graph (DAG) from the AST that avoids code duplications.

Utled og skriv ned en rettet/ein retta asyklisk graf (DAG) fra ASTen som unngår duplikasjon av kode.

6 Control Flow Graphs / kontrollflytgrafer (10 points / poeng)

The following code is given:
Se/Sja paå følgende/følgjande koden:

```
1 x = 0;
2 i = 1;
3 b = 2;
4 while (x < 100) {
5   x = x + 1;
6   if (a[x+i] > b)
7     b = x + a[i];
8 }
9 print(a[b]);
```

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

Tegn/Teikn kontrollflytgrafen (CFG) for dette kodestykket.

6.2. Draw the dependence graph for the code.

Tegn/Teikn avhengighetsgrafen/avhengnadsgrafen (*dependence graph*) for koden.

7 Three-Address Code (10 points / poeng)

The following code is given:
Se/Sja paå følgende/følgjande koden:

```
1 x = 0;
2 i = 1;
3 do {
4   if (a[x+i] > 0)
5     i = x + a[i];
6   x = x + 1;
7 } while (x < 100);
8 print(i);
```

7.1. Translate the do-loop in the code to an equivalent while-loop.

Oversett do-loopen i viste koden til en tilsvarende while-loop.

7.2. Translate the given code to three-address code (TAC). Show all intermediate steps of the translation.

Oversett den samme/same koden til tre-adressekode (TAC). Vis alle mellomskritt i oversettelsen.

7.3. When translating `switch` statements to TAC, you can use cascaded `goto`s or a jump table. Shortly discuss the advantages and differences of each approach.

Når du oversetter `switch` *statements* til TAC, kan du bruke/bruka kaskaderede `goto` instruksjoner/instruksjonar eller et/eit hoppetabell (*jump table*). Diskuter kort de/dei fordelene og forskjeller/forskjellar ved hver/kvar tilnærming.

8 Static Single Assignment Form (10 points / poeng)

The following code is given:
Se/Sja paå følgende/følgjande koden:

```
i = 1;
j = 1;
k = 0;

while (k < 100) {
  if (j < 20) {
    j = i;
    i = i + 1;
  } else {
    j = k;
    k = k + 2;
  }
  j = j + k;
}
return j;
```

8.1. Give the result of splitting the code into basic blocks.

Skriv ned resultatet av å dele/dela koden i grunnleggende/grunnleggande blokker (*basic blocks*).

8.2. Draw the control flow graph for the given code using **unique** variable names for each assignment as used in the SSA form.

Tegn/Teikn kontrollflytgrafen (CFG) for den gitte koden ved å bruke/bruka **unike** variablenavn på hver/kvar tilordning som brukt i SSA-formen.

8.3. Insert the required Phi (Φ) functions into the CFG.

Sett inn de/dei nødvendige Phi (Φ) funksjonene/funksjonane i CFG.

9 Simple Optimizations / Enkle optimaliseringer (10 points / poeng)

The following code is given:
Se/Sja paå følgende/følgjande koden:

```
int foo(n) {
  int x, y, z, a, b, c;
  x=23;
  y=42;
  z=69;

  a = x;
  a = a + 4*(x+y)*n;
  c = n;
  if (a > 10) {
    b = a * 1 + x * ( a + 0 + a);
    c = x+z;
  }
  a = b * 3;
}
```

9.1. Apply constant folding and constant propagation and give the result.

Bruk konstant folding (*constant folding*) og konstant forplantning (*constant propagation*) og skriv ned resultatet.

9.2. Apply algebraic simplification to the result of question 9.1 and give the result.

Bruk algebraisk forenkling (*algebraic simplification*) på resultatet av spørsmål 9.1 og skriv ned resultatet.

9.3. Apply strength reduction to the result of question 9.2 and give the result.

Bruk styrkereduksjon (*strength reduction*) på resultatet av spørsmål 9.2 og skriv ned resultatet.

10 lex (10 points / poeng)

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 counts the number of (English) vowels (i.e. `aeiouAEIOU`) and outputs it. Use whitespace as separators between numbers and other text.

Følgende/Følgjande lex-koden delvis implementerer en/ein skanner som slutter/sluttaer å analysere/analysera når teksten `end` blir lest/lesa. Utvid denne koden for å lage/laga en skanner/ein skannar som teller antall/tel mengde (engelske) vokaler (dvs. `aeiouAEIOU`) og skriv ut resultaten på kommandolinjen/kommandolinja. Bruk mellomrom (*white space*) som skilletegn/skiljeteikn mellom tall/tal og annen/annan tekst.

```
%{
#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;
}
```