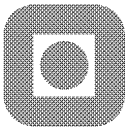
Norwegian University of Science and Technology
NTNU

Department of Computer and Information Science



FINAL EXAM [Eksamen]
TDT 4205 Compiler Technology [Kompilatorteknikk]
Wednesday, December 10, 2008
Time: 09:00-13:00

Instructional contacts [faglige kontakter] during the exam:

Anne C. Elster (918 97 062) Jan Christian Meyer (411 93 264)

Aids [hjelpemidler]: C (ingen hjelpemidler) bortsett fra ett håndskrevet notatark som er stemplet med "NTNU og Department of Computer and Information Science" som skal innleveres med alle svarene som skal være innskrevet på oppgavesettet. **Assembler syntaksen du trenger er i år skrevet inn på disse eksamensarkene.**

No aids except one NTNU stamped sheet of handwritten notes that need to be returned with this exam set along with all answers which should be given in the space provided on this exam set. This year the essential assembler syntax you need is integrated with this exam set.

[Ikkje nokre hjelpemidlar bortsett frå eit ark med handskrevne notatar som skal være stempla med "NTNU og Department of Computer & and Information Science". Dette arket skal innleverast saman med dette eksamenssettet kor du og skal gje alle svara der det er laga plass til det. I år har eg skreve assemblersyntaksen inn i disse eksamensarka for dere.]

Grades will be assigned within three working weeks.

[Karakterer vil bli satt innen tre arbeidsuker]
[Karekterar vert sette før tre airbeidsveker er omme]

It is NOT necessary to justify your answer on true/false questions.

[Det er ikke nødvendig å begrunne TRUE/FALSE spørsmål.]
[Ein treng ikkje grunngje TRUE/FALSE spørsmål.]

STUDENT NUMBER: _____

1. Warm-ups [oppvarming] – TRUE/ FALSE [Sant/u sant] (10/50 or 20% of this exam)

Circle your answers -- Note: You will get a negative score for wrong answers and 0 for not answering or circling both TRUE and FALSE.

[Sett sirkel rundt svara -- NB: Dere får negative poeng om dere svarer feil, 0 poeng for å ikke/ikkje svare eller sirkle både "TRUE"(sant) og "FALSE" (usant).]

a) The analysis part of a compiler is often called the front-end TRUE/FALSE
[Analysedelen av en/ein kompilator kalles/er og kjend som "front-end"]

b) Yacc produces tokens ["Yacc" lager "tokens"] TRUE/FALSE

c) Syntax analysis is also known as parsing TRUE/FALSE
[syntaksanalyse er det samme som parsing]

d) Semantic analyzers use the symbol table and do type checking TRUE/FALSE
[semantiske analysatorer bruker symboltabeller og utfører typesjekk]

e) Top-down parsers do not need to figure out as much of the parse tree for a given amount of input compared to bottom-up parsers TRUE/FALSE

f) Heap variables don´t explicitly occur in low-level code TRUE/FALSE
["heap"/haug variabla finnast ikke eksplisit i lav/låg-nivå kode]

g) Both Java and C allow type aliases
[Både Java og C godtar "type aliases"] TRUE/FALSE

h) new(Java/C++) or malloc (C) allocate variable space directly on stack TRUE/FALSE
[new(Java/C++) og malloc(C) alllokerer variabelplass direkte på stakken]

i) A frame (activation record) includes local variables TRUE/FALSE
[En "activation record" inkluderer lokale variabler]

j) Derived induction variables are only incremented in loop body TRUE/FALSE
[utledete induksjonvariable er kun inkrementert innen løkkene]

1. CONTINUED (Fortsettelse av Oppg. 1):

k) Type conversion may not be done automatically by the compiler TRUE/FALSE
[type-konverteringer kan ikke gjøres/gjerast av kompilatoren]

l) Static typing implies strong typing TRUE/FALSE
[Statisk typing fører til sterk typing]

m) Dataflow analysis is not part of code optimization TRUE/FALSE
[Dataflytanalyse er ikke/ikkje en/ein del av kodeoptimisering]

n) Tiling w.r.t. instruction selection involves matching tree pattern to corresponding machine instruction TRUE/FALSE
["Tiling" m.h.t. valg av instruksjoner handler om å matche tremøstre til tilsvarende maskinkode]

o) It is reasonable to assume a roughly uniform tile cost when selecting instructions on a CISC machine TRUE/FALSE
[Det er rimelig å anta at en tilnærmet uniform tiling kostnad på CISC]

p) Dominators in CFGs are used to identify loops TRUE/FALSE
["Dominators" i kontrollflytgrafer er nyttet til å identifisere løkker]

q) Tree representation may be appropriate for instruction selection TRUE/FALSE
[Tree representasjoner kan være rett for instruksjonsutvalg.]

r) The Maximal Munch algorithm finds the least expensive tiling of a DAG
["Maksimal Munch" algoritmen finner den minst dyre "tiling" av DAG]
TRUE/FALSE

s) The Pentium is a tree-address CISC architecture TRUE/FALSE
[Pentium er en tre-adresse CISC arkitektur]

t) In short-circuit code boolean operators translate into jumps TRUE/FALSE
[I "short-circuit"/kortslutnings-koder oversettes boolske operatorer til hopp]

2. **COMPILER BASICS. Fill in the blanks** (fyll inn svarene) **(8/50)**

a) What kind of information is stored in a syntax trees?
[Hva slags /Kva for type info. er lagra i et/eit syntaxtre?]

_____

b) What is a syntax-directed definition (SDD)? [Hva/Kva er SDD?]

_____

c) What is the main purpose of Dataflow analysis?
[Hva er hoved/hovud-poenget med dataflyt-analyse?]

_____

d) How is tiling used with respect to instruction selection?
[Hvordan/korleis er ”tiling” brukt/bruke vis á vis instruksjonsselektering?]

_____

e) What are Dominators in CFGs ? [Hva/Kva er”Dominators” I CFG?]

_____

f) What is backpatching? [Hva/Kva er “backpatching”?]

_____

g) What does the The Maximal Munch algorithm do?
[Hva/Kva gjør/gjer Maximal Much algoritma?]

_____

h) **Describe an example where loop unrolling would decrease performance:**
[Beskriv/Gje et/eit eksempel der utrulling av løkken vil svekke ytelsen]

3. Problem Set Related Questions [Øvingsrelaterte Oppg.] (17/50 points)

a) Related to PS1: Write a regular expression for a language of floating point constants as defined in Kernighan & Ritchie:
[Skriv et/eit regulært uttrykk for et/eit språk av flyttal konstanter som definert i Kernighan & Ritchie:]

"A floating constant consists of an integer part, a decimal point, a fraction part, an "e" or "E", an optionally signed integer exponent and an optional type suffix, one of "f", "F", "l", or "L". The integer and fraction parts both consists of a sequence of digits. Either the integer part or the fraction part (not both) may be missing: Either the decimal point or the "e" and the exponent (not both) may be missing. The type is determined by the suffix: "F" or "f" makes it float, "l" or "L" makes it long double: Otherwise it is double."

3 b) From PS 2

i) Is there a regular expression which recognizes the language of regular expressions? Explain why/why not.

ii) Is there a regular language which can be recognized by an NFA but not by any DFA? Explain why/why not.

iii) A common extension to regular expressions is backreferences, which extends their notation with the variables ($1, $2, $3, ...) to denote "the pattern which was previously matched by the (1st , 2nd , 3rd , ...) parenthesized subexpression". Does this extension affect the relationship between regular expressions and finite automata? Explain why/why not.

3 c) From PS 3: Parsing by recursive descent

Consider the grammar :

P → B
B → ET
E → DE | ε
T → TS | ε
D → tiV
V → v | ε
S → k | bBc

i) Determine which of the nonterminals are nullable, as well as their respective FIRST and FOLLOW sets.

ii) Construct a predictive parsing table for the grammar.

3 d) From PS 5:

i) Given the following grammar:

declaration → type identifier_list
type → int | float | double
identifier_list → identifier | identifier_list ', ' identifier
identifier → text

and a parser generator which only supports S-attributed grammars.
How would you associate types with identifiers?

ii) How does multiple inheritance affect method dispatch code generation?

3 e) From PS 6:

i) Create a control flow graph for the following program:

f o r (a ; b ; c) d ; e ;

3 e) continued

ii) Optimizations

```
a=1
b=2
c =3
d=a+x
e=b+c
f =e
g=f
g=d+y
a=b+c
```

In each of the cases below, state the optimization(s) applied in optimizing the above basic block.

W)

```
a=1
b=2
c =3
d=a+x
e=b+c
f =e
g=e
g=d+y
a=b+c
```

X)

```
a=1
b=2
c =3
d=a+x
e=b+c
t 1=e
f =e
g=f
g=d+y
a=t 1
```

Y)

```
a=1
b=2
c =3
d=1+x
e =5
f =5
g=5
g=d+y
a=5
```

Z)

```
a=1
b=2
c =3
d=a+x
e=b+c
f =e
g=d+y
a=b+c
```

4. Programming Question. (15/50)

The Lex/Yacc scanner/parser pair on the following pages contain the grammar of a small language of arithmetic expressions. Programs are of the form:

`eval: (expression) .[ (parameter list) ]`

where expressions are usual arithmetic expressions (including single-letter lowercase variable names). The parameter list is a comma-separated list of positive integers, which are assigned to the variable names in alphabetic order, i.e. the program

`eval:(b*b+a*a)/(10-c).[6,3,1]`

evaluates to 5 (= (3*3 + 6*6) / (10-1)).

[Scanner/Parser-paret på de påfølgende sidene inneholder grammatikken til et lite språk for aritmetiske uttrykk. Programmer tar formen

eval: (expression) .[(parameter list)]

hvor 'expression' er vanlige aritmetiske uttrykk (inklusive enkelt-tegns variabelnavn med små bokstaver). Parameterlisten er en komma-separert liste av positive heltall, som tilordnes variabelnavnene i alfabetisk orden, mao. evalueres programmet

eval:(b*b+a*a)/(10-c).[6,3,1]

til 5 (= (3*3 + 6*6) / (10-1)).]

A context-free grammar for this language is as follows:

[En kontekst-fri grammatikk for dette språket er som følger:]

program → function '[' parameter_list ']'

function → declaration eval_expression '.'

declaration → EVAL ':'

parameter_list → parameter | parameter ',' parameter_list

parameter → integer

eval_expression →

eval_expression '+' eval_expression |

eval_expression '-' eval_expression |

eval_expression '*' eval_expression |

eval_expression '/' eval_expression |

'(' eval_expression ')' |

integer |

variable

variable → VARIABLE

integer → INTEGER

The purpose of this task is to write semantic rules into the Yacc version of this grammar, to compile this language into IA-32 assembly programs with two functions: a function `eval` which represents a general translation of the expression to be evaluated, and a function `main` which pushes the []-enclosed argument list onto stack and calls `eval`. Hardcoding a list of absolute parameter values in `main` is done as a simple alternative to fetching arguments from elsewhere. The core task is to translate expressions into `eval` functions which are independent of the values they are applied to.

[Hensikten med denne oppgaven er å fylle inn semantiske regler i Yacc-versjonen av denne grammatikken, for å kompilere språket til IA-32 assemblerprogram som inneholder to funksjoner: en funksjon eval, som representerer en generell oversettelse av uttrykket som skal evalueres, og en funksjon main som skyver den []-avgrensede argumentlisten på stakk, og kaller eval. Å hardkode en liste med absolutte parameterverdier i main blir gjort som et enkelt alternativ til å hente parametre fra noe annet sted. Hovedpoenget i oppgaven er å oversette uttrykk til eval-funksjoner som er uavhengige av verdiene de anvendes på.]

Note:

- **The reason for separating the 'declaration' rule is that it will reduce before the r.h.s. of the 'function' rule, giving a semantic action in which to set up the entry point of `eval`. Similarly, the 'function' production will reduce only when the right hand side is complete, giving a natural place for emitting the code which terminates `eval`. (This is restated in the parser's comments.)**
- **The head of `main` is emitted while reducing 'parameter_list', and its tail by the reduction of 'program'. This will place it at the end of the emitted code, as the parameter list is the last syntactical element, and its completion finalizes the r.h.s. of the 'program' production.**
- **The use of single-letter lowercase variables in alphabetic order is to eliminate the need for a symbol table. You may assume not only that the assignment of parameters to variable names follows this order, but also that the use of a variable name implies that all the preceding letters are also used, resulting in a linear relationship between a variable's name and it's stack offset.**
- **The `puts` function resembles `printf`, but only accepts a literal text string (no numbers or strings may be inserted in the output using format codes).**

[Merk:

- Grunnen til å separere 'declaration'-regelen er at den dermed vil reduseres før høyre side av 'function'-regelen, for å gi plass til en semantisk regel som kan sette opp inngangen til eval. På samme vis vil 'function'-regelen først reduseres når høyre side er fullstendig, noe som gir et naturlig sted for å skrive kode for å avslutte eval. (Dette er markert i kommentarene i parserkoden.)
- Inngangen til main skrives ved reduksjon av parameter_list, og avslutningen ved reduksjon av 'program'. Dette plasserer main-funksjonen til slutt i assemblerprogrammet, ettersom parameterlisten er det siste syntaktiske elementet i et program, og fullføringen av dette avslutter høyre side av 'program'-produksjonen.
- Bruken av alfabetisk ordnede variabelnavn i enkeltvise, små bokstaver er gjort for å fjerne behovet for en symboltabell. I tråd med dette står du fritt til å anta at bruken av et variabelnavn viser at alle foregående bokstaver også er brukt, slik at det finnes et lineært forhold mellom navnet på en variabel og dens plassering på stakken.
- Puts-funksjonen ligner printf, med den forskjellen at den bare godtar rene tekststrenger (ingen tall eller strenger kan settes inn i resultatet ved hjelp av formatkoder).]

Your task is to complete the parser to perform the translation described above, by filling in strings which emit the correct assembler instructions in the puts/printf calls which are highlighted in boldface.

[Oppgaven din er å fullføre parseren slik at den utfører oversettelsen beskrevet ovenfor, ved å fylle inn strenger som skriver de korrekte assemblerinstruksjonene i puts/printf-kallene angitt i fete typer]

Small IA32 instruction reminder:

[Liten påminnelse om noen IA32-instrukser:]

movl <src>, <dst>	- move src value to dst [Flytt src til dst]
addl <src>,<dst>	- add src value to dst [Legg verdien i src til dst]
imull <src>	- multply 64-bit value in %edx:%eax by src [Gang 64-bitsverdien %edx:%eax med src]
idivl <src>	- divide %edx:%eax by src, store quotient in %eax, remainder in %edx [Divider %edx:%eax med src, plasser kvotienten i %eax, restleddet i %edx]
cltd	- sign extend %eax to %edx:%eax [Utvid fortegnet i %eax til %edx:%eax]
pushl <src>	- push src on stack [Skyv src på stakk]
popl <dst>	- pop value from stack to dst [Hent øverste verdi på stakk til dst]

Some registers and their roles: [Noen registre og rollene deres:]

%eax - results accumulator [Resultatakkumuluator]

%ebx - general data register [Generelt dataregister]

%esp - stack pointer [Stakkpeker]

%ebp - frame pointer [Rammepeker]

Scanner.l:

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%option noyywrap
%%
[ \t\n]+     {}
[0-9]+       { return INTEGER; }
eval         { return EVAL; }
[a-z]        { return VARIABLE; }
.            { return yytext[0]; }
%%
```

Parser.y:

```
%{
#include <stdio.h>
#include <string.h>
#define HEAD \
    ".section .data\n"\
    ".OUTPUT: .string \"The answer is %d\\n\"\n"\
    ".globl main\n"\
    ".text"
#define TAIL \
    "\tpushl %eax\n"\
    "\tpushl $.OUTPUT\n"\
    "\tcall printf\n"\
    "\taddl $8,%esp\n"\
    "\tleave\n"\
    "\tret"
extern char *yytext;
%}
%left '+' '-'
%left '*' '/'
%token INTEGER VARIABLE EVAL
%%
```

```
/* Emit the body of main and its return code */
program: function '[' parameter_list ']' {
        puts (
            "\tcall eval"
        );
        puts ( TAIL );
    }
    ;
/* Emit the data section, and the head of the eval function body */
declaration: EVAL ':' {
        puts ( HEAD );
          puts (




          );
    }
    ;
/* Emit the tail of the eval function body */
function: declaration eval_expression '.' {
          puts (




          );
    }
    ;
/* Emit the head of main, and place the parameter list such that it is accessible to the eval function */
parameter_list:
    parameter {
          puts (




          );
```

```
        printf (


        );
    }
    | parameter ',' parameter_list  {
        printf (


        );
    }
    ;
parameter: integer;

/* Emit the function which encodes the parsed expression */
eval_expression:
    eval_expression '+' eval_expression {
        puts (









        );
    }
    | eval_expression '-' eval_expression {
        puts (









        );
    }
```

```
    | eval_expression '*' eval_expression {
        puts (
        );
    }
    | eval_expression '/' eval_expression {
        puts (
        );
    }
    | '(' eval_expression ')'
    | integer  {
        printf (
        );
    }
    | variable {
        printf (
        );
    } ;
/* Terminals - convert text to integer and variable name to alphabetic index */
variable: VARIABLE  { $$ = yytext[0] - 'a'; } ;
integer: INTEGER    { $$ = strtol ( yytext, NULL, 10 ); } ;
%%
int yyerror ( void ) { puts ( "Syntax error" ); return 1; }
int main ( int argc, char **argv ) { yyparse(); }
```