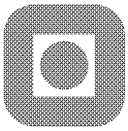
Norwegian University of Science and Technology
NTNU

Department of Computer and Information Science



FINAL EXAM [Eksamen]
TDT 4205 Compiler Technology [Kompilatorteknikk]
Tuesday, June 8, 2010
Time: 09:00-13:00

Instructional contacts [faglige kontakter] during the exam:

Jan Christian Meyer (mob. 41 19 32 64 kontor 73 59 05 42)

Aids [hjelpemidler]: C (ingen hjelpemidler). Alle svar skal være innskrevet på oppgavesettet. Assemblersyntaksen du trenger er skrevet inn på disse eksamensarkene.

C (ingen hjelpemidler). Alle svar skal vere skrivne på oppgåvesettet. Assemblersyntaksen du treng er skriven inn på desse eksamensarka.

C (No aids). 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.

Grades will be assigned within three working weeks.

[Karakterer vil bli satt innen tre arbeidsuker /
Karakterar vil bli satt innan tre arbeidsveker]

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

[Det er ikke nødvendig å begrunne TRUE/FALSE spørsmål. /

Det er ikkje naudsynt å grunngje TRUE/FALSE spørsmål.]

STUDENT NUMBER: _____

1 True / False [Sant/usant] (10 / 50)

a) LR parsing produces a rightmost derivation.
[LR parsing produserer en/ein derivasjon fra/frå høyre/høgre (rightmost)]
TRUE / FALSE

b) Interfaces (as in Java) specify a dispatch vector layout.
[Interface (som i Java) spesifiserer utlegget til en dispatchvektor] TRUE / FALSE

c) An SLR grammar can not be ambiguous.
[En/ein SLR-grammatikk kan ikke være/ikkje vere tvetydig] TRUE / FALSE

d) It is possible to write a regular expression for the language of regular expressions.
[Det er mulig/mogeleg å skrive et/eit regulært uttrykk for språket av regulære uttrykk]
TRUE / FALSE

e) The execution cost of a load instruction is always the same.
*[Kostnaden ved å utføre en/ein load-instruks er alltid den samme/same]*TRUE / FALSE

f) A sound type system may reject a type-safe program.
[Et/eit sunt (sound) typesystem kan avvise et/eit typesikkert program] TRUE / FALSE

g) The x86 assembler takes 3-address code as input.
[x86-assembleren tar 3-addressekode som input] TRUE / FALSE

h) Reaching definitions is a backward analysis.
[Reaching definitions er en/ein bakover-analyse] TRUE / FALSE

i) Adding lookahead symbols increases the size of the corresponding parsing table.
*[Ytterligere lookahead-symboler øker størrelsen på parsetabellen /
Ytterlegare lookahead-symbol aukar storleiken på parsetabellen]*
TRUE / FALSE

j) The handle of a right-sentential form is also called a viable prefix.
*[Et annet navn på en right-sentential setningsform er 'viable prefix' /
Eit anna namn på ein right-sentential setningsform er 'viable prefix']* TRUE / FALSE

k) Reduce/reduce conflicts can commonly be resolved by adding precedence rules to the parsing scheme.
*[Reduce/reduce-konflikter kan vanligvis avklares ved å innføre presedensregler i parsingmetoden /
Reduce/reduce-konflikter kan vanlegvis avklarast ved å innføre presedensreglar i parsingmetoden]* TRUE / FALSE

l) Strong type checking depends on a static type system.
[Sterk typesjekking er avhengig av et/eit statisk typesystem] TRUE / FALSE

m) In a control flow graph, the condition of a loop dominates its body.
[I en/ein kontrollflytgraf dominerer (dominate) betingelsen/vilkåret i ei løkke koden inne i løkka] TRUE / FALSE

n) It is possible to write a context-free grammar for the language of context-free grammars.
[Det er mulig/mogeleg å skrive en/ein kontekstfri grammatikk for språket av kontekstfrie grammatikker/grammatikkar] TRUE / FALSE

o) Every pair of elements in a partial order must be comparable by the ordering relation.
[Hvert elementpar i en partiell orden må være sammenlignbart ihht. den ordnende relasjonen /
Kvart elementpar i ein partiell orden må vere samanliknbart ihht. den orndnande relasjonen] TRUE / FALSE

p) When a function returns, any values it allocated on heap are no longer valid.
[Når en/ein funksjon returnerer, er verdier/verdiar den allokerte på heap ikke/ikkje lenger gyldige] TRUE/ FALSE

q) Left factoring a grammar reduces the number of lookahead symbols required to parse it predictively.
[Venstrefaktorisering av en/ein grammatikk reduserer antall symboler/symbol som må leses/lesast som lookahead for å parse den prediktitivt] TRUE / FALSE

r) When a recursive descent parser verifies any predicted terminal, it already knows which lexeme to expect.
[Når en recursive descent-parser bekrefter en forutsagt terminal, vet den alt hvilket leksem som er forventet /
Når ein recursive-descent-parser bekrefter ein forutsagt terminal, veit den alt kva for eitt leksem som er forventa] TRUE / FALSE

s) With short-circuit evaluation, binary boolean operators can translate into jumps.
[Med short-circuit evaluering kan binære boolske operatorer/operatorar oversettes/oversettjast til hopp] TRUE / FALSE

t) The set of languages recognizable by LALR(k) parsers is smaller than that of LR(k) parsers.
[Mengden språk som kan gjenkjennes av LALR(k)-parsere er mindre enn den som kan gjenkjennes av LR(k)-parsere /
Mengda av språk som kan gjenkjennast av LALR(k)-parsere er mindre enn den som kan gjenkjennast av LR(k)-parsere] TRUE / FALSE

Short answers [Korte svar] (10/50)

1. Briefly explain the distinction between L-attributed and S-attributed syntax-directed definitions. ***[Gi en/ei kort forklaring på distinksjonen mellom L-attributed og S-attributed syntaksrettede/syntaksretta definisjoner/definisjonar]***

_____

_____

2 Under what condition is the maximal fixed point solution to a dataflow problem identical to the meet-over-paths solution?
[Under hvilken betingselse er maksimal-fikspunkt-løsningen på et dataflytproblem identisk med meet-over-paths-løsningen? /
Under kva for vilkår er maksimal-fikspunkt-løsningen på et dataflytproblem identisk med meed-over-paths-løysinga?]

_____

_____

3 How is the ε-closure of an NFA state defined?

[Hvordan er ε-tillukningen (ε-closure) definert for en NFA-tilstand? /
Korleis er er ε-tillukninga (ε-closure) definert for ein NFA-tilstand?]

_____

_____

4 Can a language with pointer arithmetic feature automatic garbage collection? Justify your answer. ***[Kan et/eit språk med pekeraritmetikk / peikeraritmetikk også ha automatisk garbage collection? Begrunn/grunngje svaret.]***

_____

_____

_____

5 State two advantages obtained by function inlining. ***[Oppgi to fordeler som oppnås ved inlining av funksjoner. / Oppgje to fordelar som ein oppnår ved inlining av funksjonar.]***

_____

_____

6 How can register assignment benefit from live variable analysis?

[Hvordan/korleis kan registertilordning dra nytte av live variable-analyse?]

_____

_____

_____

7 Most programs exhibit some degree of spatial and temporal locality. Explain what these two terms refer to, and how they may be exploited for efficient execution. ***[De fleste programmer har i noen grad romlig (spatial) og temporal lokalitet. Forklar hva disse begrepene innebærer, og hvordan de kan utnyttes for effektiv programutføring. /***
Dei fleste program har i nokon grad romleg (spatial) og temporal lokalitet. Forklar kva desse omgrepa inneber, og korleis dei kan utnyttjast for effektiv programutføring.]

_____

_____

_____

_____

_____

8 LL(k) parsing can be extended to an unbounded amount of lookahead by allowing the parser to decide the choice of production based on testing the remaining token stream against a finite set of regular languages. Does this resolve the problem with left-recursion? Explain.
[LL(k)-parsing kan utvides til ubegrenset lookahead ved å la parseren bestemme valget av produksjon basert på å teste utestående tokenstrøm mot en endelig mengde regulære språk. Ordner dette problemet med venstrerekursivitet? Forklar. /
LL(k)-parsing kan utvidast til ubegrensa lookahead ved å la parseren bestemme valet av produksjon basert på å teste utestående tokenstraum mot ei endeleg mengde regulære språk. Ordnar dette problemet med venstrerekursivitet? Forklar.]

_____

_____

_____

_____

_____

9 Explain the cause and effect of heap memory fragmentation.

[Forklar årsaken til, og virkningen/verknaden av fragmentering av heap-minne.]

10 Identify the loop-invariant code in the following program fragment, and explain the benefit of moving it. ***[Finn den løkke-invariante koden i det følgende/følgjande programfragmentet, og forklar fordelen ved å flytte den.]***

```
double s[180], pi = M_PI;

for ( int d=0; d<180; d++ )

    s[d] = sin ( d * pi / 180.0 );
```

Miscellaneous [Diverse] (15/50)

a) Consider the simple grammar: ***[Betrakt følgende/følgjande enkle grammatikk:]***

S → XxXy | YyYx

X → ε

Y → ε

Show the LL(1) parsing table ***[Vis LL(1)-parsetabellen]***

Write out the steps in a top-down parse of the input 'xy', showing remaining input, and the state of the parse stack for each step.
[Skriv ut stegene i en ovenfra/ned-parsing av input 'xy'. Vis utestående input, og tilstanden til parsestakken for hvert steg. /
Skriv ut stega i ein ovanfra/ned-parsing av input 'xy'. Vis uteståande input, og tilstanden til parsestakken for kvart steg.]

Write out the steps in a bottom-up parse of the input 'xy', showing remaining input, state of the parse stack, and the action taken for each step.
[Skriv ut stegene i en nedenfra/opp-parsing av input 'xy'. Vis utestående input, tilstanden til parsestakken, og handling (action) utført i hvert steg. /
Skriv ut stega i ein nedanfra/opp-parsing av input 'xy'. Vis uteståande input, tilstanden til parsestakken, og handling (action) utført i kvart steg.]

b) Draw the layout of a process image at run time, labelling each part.
[Tegn utlegget til en prosess ved kjøretid, med delene merket /
Teikn utlegget til ein prosess ved køyretid, med delane merka]

c) Draw the layout of an activation record, labelling each part.
[Tegn utlegget til en activation record, med delene merket /
Teikn utlegget til ein activation record, med delane merka]

d) Briefly describe which changes you would make to your compiler from the TDT4205 problem sets, in order to extend VSL with global variables.
[Gi en kort beskrivelse av hvilke endringer du ville gjøre i kompilatoren fra øvingsopplegget i TDT4205, for å utvide VSL med globale variabler /
Gi ein kort skildring av kva endringar du ville gjere i kompilatoren frå øvingsopplegget I TDT4205, for å utvide VSL med globale variablar]

Programming [Programmering] (15/50)

The lex specification on the following pages is a partially completed recursive descent parser for the grammar

program → def *name expression program* | *expression*

expression → (*expression expression name*) | integer | A | B

name → identifier

where the terminals 'integer', 'identifier' are the tokens of (signed) integers, unbroken lowercase strings, and 'def', 'A' and 'B' stand for their literal strings.

[Lex-spesifikasjonen på de/dei følgende/følgjande sidene er en/ein delvis fullført recursive-descent-parser for grammatikken ovenfor, hvor terminalene 'integer' og 'identifier' er tokens for heltall/heiltal (m. fortegn) og ubrutte/ubrotne strenger av små bokstaver. 'def', 'A' og 'B' står for tekstliteraler.]

The program is intended to compile a simple language of binary expressions in postfix notation, by using function calls (with two parameters) for operations. The functions 'add', 'sub', 'mul' and 'div' are defined in the (incomplete) TEXT_HEAD macro printed at the beginning of output. Each operation (function) will have exactly two operands (parameters), which are referred to by the literals A and B.

[Programmet er tenkt til å kompilere et enkelt språk av binære uttrykk i postfiksnotasjon, ved bruk av funksjonskall (med to parametre) som operasjoner. Funksjonene 'add', 'sub', 'mul' og 'div' er definert i den (ufullstendige) makroen TEXT_HEAD, som skrives foran/fremst i output. Hver/kvar operasjon (funksjon) skal ha nøyaktig to operander (parametre), som henvises til / blir viste til av literalene A og B.]

As an example, a program like

```
def dprod ( (A A mul) (B B mul) add )
( 4 2 dprod )
```

Should

- generate the function 'dprod' by printing appropriate calls to 'mul' and 'add'
- evaluate the expression which comes from calling 'dprod' with parameters A=4 and B=2. (This example evaluates to 4*4+2*2 = 20)

[Eksempelvis/til dømes skal programmet ovenfor/ovanfor

- *generere funksjonen 'dprod' ved å skrive ut passende kall til 'mul' og 'add'*
- *evaluere uttrykket som kommer/kjem av å kalle 'dprod' med parametre A=4 og B=2. (Eksempelet/dømet evaluerer til 4*4+2*2 = 20)]*

The 'traverse' function (p.14) handles nonterminals, and the traversal of the implicit syntax tree. The 'verify' function on (p.15) prints any relevant code for terminals, and advances input.

[Funksjonen 'traverse' (s.14) håndterer/handterar nonterminaler, og traverseringen av det implisitte syntakstreet. Funksjonen 'verify' (s.15) skriver/skriv ut relevant kode for terminaler/terminalar, og rykker input framover.]

The call mechanism is given by the 'NAME' clause in the 'verify' function, which prints a call to the named function, and substitutes the top 2 values on stack for the result in EAX on return. The order of evaluation is given, as the left operand (A) will evaluate before the right (B), and therefore be found below it on stack.

[Kallmekanismen følger av 'NAME'-klausulen i 'verify'-funksjonen, som skriver ut et kall på den navngitte funksjonen, og erstatter øverste 2 verdier på stakk med resultatet i EAX ved retur. Evalueringsorden er gitt, fordi den venstre operanden (A) evalueres før den høyre (B), og derfor vil finnes nedenfor den på stakk.]

(The macros MAIN_HEAD and MAIN_TAIL referred to in the 'program' production are not included, you may assume that they contain code to read two integers from the command line and push them into the positions of A and B before the code which evaluates the final expression is printed.)

[(Makroene MAIN_HEAD og MAIN_TAIL som brukes i 'program'-produksjonen er ikke inkludert, du kan anta at de inneholder kode som leser to heltall fra kommandolinjen, og skyver dem på plassene til A og B før koden som evaluerer det siste uttrykket skrives ut.)]

Your task is to

1. complete the 4 built-in functions in TEXT_HEAD so that the correct IA32 assembly is printed. (Note that since each of the operands appears only once, and at the beginning of such expressions, it is not necessary to set up a stack frame for these)
2. complete the DEF production, so that it prints a label and activation record, parses the expression, and emits code to return the result in EAX
3. complete the EXPRESSION production, so that it correctly parses expressions and prints code (using the 'verify' function)
4. complete the cases for terminals A and B in the 'verify' function

[Oppgaven/oppgåva er å

1. *fullføre de 4 innebygde funksjonene i TEXT_HEAD, slik at korrekt IA32 assembly skrives/blir skrive ut. (Merk at siden operandene opptrer bare en gang, først i slike uttrykk, er det ikke nødvendig å sette opp en stakkramme for disse.)*
2. *fullføre DEF-produksjonen, slik at den skriver/skriv ut en label og activation record, parser uttrykket, og returnerer resultatet i EAX*
3. *fullføre EXPRESSION-produksjonen, slik at den parser uttrykk korrekt, og skriver ut kode (vha. 'verify'-funksjonen')*
4. *fullføre case-setningene for terminalene A og B i 'verify'-funksjonen*

]

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/øvste 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 / stakkpeikar]

`%ebp` – frame pointer [Rammepeker / rammepeikar]

```
/* This is the definitions section: it's primary purpose is to define a
   macro which contains the text of the 4 predefined functions.
   Additionally, it defines an enumeration of the grammar symbols, and a
   buffer for the lookahead symbol.
 */
%{
#include "system.h"
#define TEXT_HEAD \
".data                          \n" \
".OUT: .string \"%%d\\n\"       \n" \
".globl main                    \n" \
".text                          \n" \
"add:                           \n" \

"_______________________________\n" \

"_______________________________\n" \

"_______________________________\n" \

"sub:                           \n" \

"_______________________________\n" \

"_______________________________\n" \

"_______________________________\n" \

"mul:                           \n" \

"_______________________________\n" \

"_______________________________\n" \

"_______________________________\n" \

"_______________________________\n" \
"div:                           \n" \

"_______________________________\n" \

"_______________________________\n" \

"_______________________________\n" \

"_______________________________\n"

typedef enum {
    PROGRAM, EXPRESSION,    // Non-terminals
    NAME, INTEGER, DEF, A, B            // Terminals
} symbol_t;
symbol_t lookahead;

void quit ( void )
{
    fprintf ( stderr, "Syntax error\n" );
    exit ( EXIT_FAILURE );
}
%}
%option array
%option noyywrap
%option yylineno
```

```
/* The following is the rules section which generates yylex() to tokenize
   the program, prototypes for the traverse and verify functions, and a
   main function which reads the first token and starts traversal.
 */

%%
[\ \t\n]+   {}
def         { return DEF; }
[a-z]+      { return NAME; }
A           { return A; }
B           { return B; }
-?[0-9]+    { return INTEGER; }
.           { return yytext[0]; }
%%


void traverse ( symbol_t nonterm );
void verify ( symbol_t expected );


int
main ( int argc, char **argv )
{
    printf ( TEXT_HEAD );
    lookahead = yylex();
    traverse ( PROGRAM );
    exit ( EXIT_SUCCESS );
}
```

```
/* This is the 'traverse' function, which embodies the recursive traversal
    of the syntax tree. It relies on the verify function to print
    appropriate code when a terminal is encountered.
 */
void
traverse ( symbol_t nonterm )
{
    switch ( nonterm )
    {
        case PROGRAM:
            switch ( lookahead )
            {
/* No more definitions, print the code to evaluate the final expression */
                case '(': case INTEGER: case A: case B:
                    printf ( MAIN_HEAD );
                    traverse ( EXPRESSION );
                    printf ( MAIN_TAIL );
                    break;

                case DEF:
                    verify ( DEF );       /* Verify the 'def' terminal */
/* Print label, set up activation record, parse expr, take down record,
    return */









/* Continue traversal - another DEF, or the expression to evaluate */
                    traverse ( PROGRAM );
                    break;
            }
            break;
        case EXPRESSION:
            switch ( lookahead )
            {
                case '(':
/* Handle the traversal of a parenthesized expression */









                    break;
                case INTEGER: case A: case B:
/* Terminals - these need only be verified (and have any code printed) */
                    verify ( lookahead );
                    break;
            }
            break;
    }
}
```

```
/* The following is the 'verify' function. Its primary purpose is to verify
    that a token matches what is expected at a point in the traversal, and
    advance the input / token stream. Secondarily, tokens representing
    terminals which require code to be printed are detected, and
    appropriate strings are output.
 */
void
verify ( symbol_t expected )
{
    if ( lookahead == expected )
    {
        switch ( expected )
        {
            case INTEGER:
                printf ( "    pushl $%s     \n", yytext );
                break;

            case A:
/* Get the value of the bottom (first) parameter */



                break;

            case B:
/* Get the value of the top (second) parameter */



                break;

            case NAME:
/* A name prints a call to the named function, and substitutes the top 2
    values on stack with the result from EAX */
                printf ( "    call %s\n", yytext );
                printf (
                    "    addl $8,%%esp  \n"
                    "    pushl %%eax  \n"
                );
                break;
        }
        lookahead = yylex();
    }
    else
        quit();
}
```