

# Runtime Code Generation

Kristian Dupont Knudsen <kristian@chrylers.com>  
Vejleder: Torben Ægidius Mogensen

---

# Runtime Code Generation

by Kristian Dupont Knudsen

Vejleder: Torben Ægidius Mogensen

Copyright © 2005 Kristian Dupont

Tekst og kode til denne rapport findes her: [/net/skuld/home/disk24/di040508/RTCG](#) og her: <http://chrylers.com/software/rtcg.zip>.  
Teksten er skrevet i docbook xml.

---

---

---

---

# Table of Contents

Indledning .....	vi
1. C++ og multi-stage programmering .....	1
1.1. Hvorfor c++? .....	1
1.2. Template metaprogrammering og partial evaluation .....	1
1.3. STL functional .....	1
2. Et dynamisk framework .....	3
2.1. Fremgangsmåder .....	3
2.2. Intel IA32 platformen .....	3
2.2.1. Lidt historie .....	3
2.2.2. En assembler i C++ .....	3
2.3. Dynamiske konstruktioner .....	5
2.3.1. Opbygning og syntaks .....	5
2.3.2. Implementation .....	7
3. I praksis.. .....	10
3.1. Message Dispatching .....	10
3.2. Bitblt .....	12
3.3. STL functional .....	12
4. Afprøvning .....	14
5. Fremtidige milepæle og andre systemer .....	17
6. Konklusion .....	19
Referencer .....	20
A. Kildekode .....	21

---

## List of Tables

1. Kodegenereringsteknikker .....	vi
-----------------------------------	----

---

# Indledning

Moore's lov har for nyligt haft 40 års jubilæum. Hardware udvikler sig med forrygende hast og det vil sandsynligvis fortsætte et stykke tid endnu. Al denne ydelse har givet programmører et pusterum hvilket har skabt en række fordele for software design. Man kan med god samvittighed skrive kode i et fortolket sprog der kun bruger en brøkdel af cpukraften på det som programmøren egentlig beder den om. Man kan lave abstraktioner og systemer som virtuel hukommelse og garbage collection der hjælper programmørerne med at fokusere på det store billede, imod betaling af en del ydelse. Generelt kan man tale om en tendens til at vi arbejder med stadig mere abstrakte og deklarative systemer. Dette giver god mening da det er den eneste måde man kan udvikle stadig større arkitekturer.

Og hvordan kan man så forsvare at arbejde med noget så modstridende som dynamisk kodegenerering på kørselstidspunktet, der netop giver kompleksitet i koden mod en gevinst i ydelse? Det giver mening fordi at på trods af denne udvikling er det stadigvæk meget få løkker der dominerer det samlede regnskab over hvad cpu'en bruger tid på. Disse løkker kan det betale sig at optimere på.

Mit mål er at gøre det muligt at sammensætte kode under kørslen, kendt som *run-time code generation*, således at man kan nøjes med den kode som er nødvendig. På kørselstidspunktet har man en meget større viden om hvad der er nødvendigt, idet en delmængde af de implicerede variable vil være kendte. Hvis man skulle sætte det i hierarki med andre former for kodegenerering, vil man kunne opstille en tabel lignende denne:

**Table 1. Kodegenereringsteknikker**

Teknik	Kodegenetaorens involvering	Beskrivelse
Statisk oversættelse	Én gang	Kun statisk data kendes
JIT - Just In Time compilation	Én gang pr afvikling af programmet	Viden om aktuel CPU og anden information kan udnyttes
<i>RTCG - Run-Time Code Generation</i>	Mange gange pr afvikling af programmet	Viden om kontekst i form af variable og andet kan udnyttes
Fortolkning	Hvert udtryk behandles under afvikling af programmet	Meget specifik viden om den aktuelle kontekst kan udnyttes

I hierarkiet ligger RTCG som det punkt der involverer kodegenerering næstmest - kun overgået af fortolkning. JIT eller Just In Time compilation, der er blevet gjort populært af Java og .NET i nyere tid, genererer kode umiddelbart før det skal afvikles, men *i reglen* kun én gang - når koden er blevet genereret bliver den afviklet herfra igen næste gang den kaldes. Dette gør at en JIT oversætter vil være forsigtig med at binde variable der kan tænkes at variere hvor man med RTGC kan vælge at oversætte gentagne gange, hver gang med nye variable. Det åbner muligheden for at vinde meget ydelse de små steder hvor det virkelig tæller. Fortolkning er endnu mere ekstrem idet det involverer kodegenerering for hvert udtryk, hele tiden - i hvert fald i nogle udgaver. Her kunne man i teorien opnå meget stor ydelse, men fortolkningen sænkes ofte af selve kodegeneratoren, der hele tiden ligger aktiv bagved. Hvis man vil opnå den optimale ydelse er run-time code generation altså et ret interessant område.

Målet med denne opgave er at beskrive en løsningsmodel for dette problem - ikke en komplet løsning, da en sådan vil være meget omfattende. Jeg vil vise hvordan man kan lave et system i C++ der er i stand til at skalere til at kunne bruges i virkelige projekter af forskellig art. I denne første, simple udgave af koden vil jeg ikke fokusere på gængse optimeringsteknikker som en almindelig oversætter benytter sig af. Det vil sige at den kode, der kommer ud kun udmærker sig ved at være dynamisk konstrueret hvilket i sig selv kan give nogle ydelsesfordele, men ikke optimeret i øvrigt, hverken hvad angår hastighed eller hukommelsesforbrug.

---

# Chapter 1. C++ og multi-stage programmering

## 1.1. Hvorfor c++?

Diskussioner for og imod programmeringssprog har en tendens til at blive "religiøse" af natur, dvs. argumenterne baserer sig gerne på personlig smag snarere end en objektiv vurdering. Jeg skal forsøge at undgå dette. Et oplagt argument for at vælge C++ til en type af opgave som denne er, at det er meget udbredt at den ydelseskrevende del af et system er skrevet i C++. Hvis man eksempelvis vil skrive et bibliotek til det populære sprog Python, er det meget almindeligt at skrive et bibliotek i C++ som så enten wraps med Boost::Python<sup>1</sup> eller måske med SWIG<sup>2</sup> - begge biblioteker der gør det let at tilgå C++ kode fra fortolkede sprog. Så om ikke andet, må C++'s popularitet være det afgørende for mit valg.

## 1.2. Template metaprogrammering og partial evaluation

Generisk kode er noget vi alle som programmører søger at opnå. Jo mere den samme stump kode kan bruges til, jo mindre bliver man nødt til at duplikere den og jo lettere bliver det at holde styr på den samlede, voksende mængde kode. Problemet med generisk kode er, at det i reglen vil indeholde en mængde logik der ikke benyttes i givne situationer. Hvis én funktion skal kunne håndtere to situationer af en eller anden art, vil man som regel indføre betinget udførelse i form af if-sætninger og lignende.

Dette problem kan man løse med partiel evaluering, som DIKU i øvrigt er kendt for at specialisere sig i. Idéen er, at generisk kode specialiseres med et antal kendte variable hvilket resulterer i ny kode der er mindre generisk men som leverer bedre ydelse og som fylder mindre.

Todd Veldhuizen og andre har udforsket brugen af C++ templates som partiel evaluator. Templates, viser det sig, udgør et Turing-fuldstændigt programmeringssprog i sig selv. Dette opdagede Erwin Unruh, og han skrev et kort program der i oversætterens output udskriver printal. Denne teknik kan med fordel benyttes hvis man for eksempel skriver et bibliotek der skal kunne bruges i mange henseender. Det giver en fremragende genericitet, uden at kompromittere ydelsen. Der er dog et uheldigt krav her: man skal kende de variable man vil specialisere ud fra på oversættelsestidspunktet. Man kan få oversætteren til at skrive mange udgaver af samme kode, specialiseret på forskellig vis, i samme program og så vælge mellem disse under afvikling, men dette kan resultere i endog meget store programmer, lange oversættelsestider osv.

Metaprogrammering ved hjælp af templates er altså ikke i sig selv en optimal løsning på vores problem.

## 1.3. STL functional

En meget central del af standard biblioteket til C++ er STL, eller Standard Template Library. Dette bibliotek benytter en kombination af klasser, funktioner og templates til at levere et bibliotek der er meget inspireret af funktional programmering. Det præsenterer en række datastrukturer og algoritmer, adskilt af en helt ren snitflade, der samtidig passer sammen med de traditionelle C idiomer som arrays og iteration via pointere. STL har den tydelige fordel at det er standard C++ og dermed i teorien er tilgængeligt for enhver C++ programmør. Det eneste, der kræves er en oversætter der opfylder standardsepcifikationen for C++ (hvilket dog er lidt besværligt - faktisk findes der i øjeblikket ingen oversætter der fuldt ud

---

<sup>1</sup> Boost::Python [<http://www.boost.org/libs/python>]

<sup>2</sup> SWIG [<http://www.swig.org>]

understøtter den standard der blev skrevet i 1998! I de senere år er situationen dog blevet væsentligt forbedret og selv Microsoft er begyndt at kunne følge med).

STL indeholder et antal funktioner og klasser der har til henblik at tillade en art funktionel programmering i C++. Med disse kan man, med en lidt omstændig syntaks, konstruere funktioner af højere orden, sammensatte samt anonyme funktioner. Her bliver det interessant, for funktionerne `bindxxx` tillader for eksempel at man kan skrive følgende:

```
// Generel potensfunktion, der returnerer x^exponent.
float power(float x, int exponent)
{
    float result = x;
    for(int i = 1; i != exponent; ++i)
        result *= x;
    return result;
}
// ...
{
    // konstruér en specialiseret udgave af power, power2,
    // der har eksponenten 2.
    function<float (int)> power2 = bind(&power, _1, 2);

    float x = 2.0;
    float result = power2(x);
    cout << result << endl;           // Vil udskrive 4.
}
```

3

Dette præsenterer tilsyneladende en løsning på vort problem: med dette kan man lave funktioner af højere orden, der producerer specialiserede funktioner under kørsel. Desværre vokser træerne ikke ind i himlen. Selv om det abstrakt synes at være en løsning, er implementationen det ikke. Det, der skabes bag kulisserne, er en såkaldt functor, eller et function object. Det udnytter C++'s mulighed for at overskrive operatorer - herunder anvendelsesoperatoren (), der kan få en klasse til at opføre sig som en funktion. Med en sådan kan man sammenhænge data og kode - man kan gemme værdi i variable og gemme en pointer til en funktion som man kan kalde med disse variable som parametre. Semantisk er dette identisk med en specialiseret funktion, men desværre har man ikke de ydelsesmæssige fordele der ville være ved at have en rigtig specialiseret funktion. Tværtimod - ud over at den oprindelige kode stadig gennemløbes, er der blevet tilføjet et funktionskald til sekvensen idét man først kalder functor'ens anvendelsesoperator (der er en funktion i sig selv), som derefter kalder den oprindelige funktion med de parametre som den har lagret.

Vi kan dog drage nytte af denne mulighed for at abstrahere funktionskald ud i data, som vi senere skal se.

<sup>3</sup> Bemærk at dette er skrevet med en version af `bind` der ikke findes i standard biblioteket i skrivende stund, men derimod i biblioteket `Boost::bind` [<http://www.boost.org/libs/bind/bind.html>], der er en udvidelse som forventes optaget i den kommende standard, samt `Boost::function` [<http://www.boost.org/libs/function>], en tilsvarende udvidelse.

---

# Chapter 2. Et dynamisk framework

## 2.1. Fremgangsmåder

Jeg vil forsøge at konstruere et framework, dvs. et klasse/funktionsbibliotek der kan sætte en programmør i stand til at konstruere ægte dynamisk genereret kode under kørslen. Man skal på én eller anden måde kunne specificere hvordan et funktionskald skal se ud. Derefter vil man kunne afvikle sådan et i for eksempel en løkke der gennemløbes mange gange og dermed have optimeret ydelse. For at kunne opnå dette bliver jeg nødt til at forbyde mig imod god programmørskik - jeg bliver nødt til at skrive kode der er platformsafhængig. Da der ikke er nogen form for fortolkning eller oversættelse til stede i et c++ program efter det er oversat, bliver jeg nødt til at lave det selv, og hvis jeg skal lave noget der kører hurtigt vil platformsuafhængige løsninger involvere alt for mange betingede spring, funktionskald osv. Min løsning er at samle maskinkode for på den måde at skabe en funktion der kan kaldes. Da jeg selv sidder ved en PC med Intel processor i og det i øvrigt er den cpu jeg har mest kendskab til, vil jeg bruge den som udgangspunkt.

Min tilgang til problemet er bottom-up af natur. Man kunne vælge at implementere et programmeringsproglignende system, der opstiller syntakstræer og genererer kode herudfra. Jeg har valgt i stedet at give brugeren mulighed for at sammensætte kode ved hjælp af en simpel assembler. Denne kan så lægges ind i større klasser og funktioner for at stille mere kompleks funktionalitet til rådighed. Denne fremgang gør at man hurtigt kan komme i gang - til gengæld er den sikkert mere besværlig at portere til andre platforme end den mere abstrakte ville være.

## 2.2. Intel IA32 platformen

### 2.2.1. Lidt historie

IA32 cpu'erne, også kendt som x86 familien, har en last af bagudkompatibilitet at slæbe rundt på. Med rødder der kan anes helt tilbage i gamle Intel chips som 8008 og 4004, understøtter de mange instruktioner der tidligere blev betragtet som smarte, men som man sidenhen har indset oftest gør mere skade end gavn. Der er især tale om et antal komplekse instruktioner, der i praksis hver kan erstattes af en række mindre og simplere instruktioner (hvilket også sker i en mikrosimulering i de nyere processorer). Af samme grund vil jeg ikke implementere hele instruksionssettet i assembleren, da dette vil være omsonst og blot tilbyde en masse funktionalitet som det ikke kan betale sig at bruge.

Et større problem ved bagudkompatibiliteten er det rod som instruksionssettet er blevet til bag facaden. Eftersom det har gennemlevet en glidende overgang fra 16 til 32 bit hukommelsesadressering, findes alle instruktioner i 16 og 32 bit udgaver. Disse instruktioner manipulerer de samme registre, hvor 16bit udgavene simpelthen arbejder med den ene halvdel af de tilsvarende 32bit registre. Dertil kommer at arkitekturen er af mere imperativ natur end funktionel - man vedligeholder tilstand. Faktisk ligger det tydeligt i registernavnene "accumulator" og "count" registre er beregnet til at holde information der ændres løbende - hvorimod en RISC processor for eksempel har mere funktionelle instruktioner der tager to registre og placerer resultatet i et tredje. Disse ting, samt at IA32 har meget få registre i det hele taget, gør det til en god gammeldags udfordring at skrive god kode uden en masse hukommelsestilgang der som bekendt er meget ineffektivt i forhold til databehandling i selve registrene.

### 2.2.2. En assembler i C++

Maskinkode er meget simpelt at producere. En helt dum assembler skal ikke kunne meget andet end at sætte bytes efter hinanden i et reserveret område i hukommelsen. Når man så vil afvikle koden sætter man bare instruksionspointeren til at pege på begyndelsen af dette område. Der er dog en enkelt ting man skal holde styr på, nemlig labels og spring i koden.

Man kan vælge at implementere en assembler på mange forskellige måder. Formålet er at man på én eller anden måde kan komme fra en række specificeringer af instruktioner til et array af bytes med maskinkoden i. Dette kunne man for eksempel lave ved at lade brugeren angive assemblerkode i form af strenge. Strenge har den fordel at de qua deres uafhængighed fra oversætteren kan indeholde hvad som helst. Dermed kan man tillade en syntaks der er magen til kendte assemblere, og derved gøre det lettere for brugeren. (Måske kunne man tilmed bruge kode fra en allerede eksisterende assembler). Man kunne forestille sig kode der lignede dette:

```
assembler.assemble("mov eax, 100");
```

Dette svarer nøjagtig til MASM syntaksen som er meget anvendt. Det vil også virke fint, men efter min mening ikke optimalt. Med det vil man ligeledes kunne skrive følgende:

```
assembler.assemble("Bispens gipsgebis");
```

Dette program vil give en fejlmeddelelse af en eller anden art, men først når programmet køres. Selv om projektet her måske kan give det indtryk at jeg er for run-time-aling, er det ikke tilfældet med fejlmeddelelser. Det er klart at vi ikke med alverdens teknologi kan forhindre folk i at skrive kode med fejl i, men en fejl som den illustrerede (og andre, mere subtile) vil kunne fanges allerede på oversættelsestidspunktet hvis vi tager C++'s typesystem til hjælp. Dette skal ikke forstås som om at mit mål er at lave en typechecked assembler, for det er ikke tilfældet. Hvad man placerer i sine registre skal man selv holde styr på. Det, vi kan checke via C++ typesystemet er, at man ikke forsøger at konstruere ulovlige eller ikke-eksisterende instruktioner.

Mark Hopkins har i en fremragende oversigt <sup>4</sup> der så vidt jeg ved kun er blevet publiceret på nyhedsgruppen `alt.lang.asm`, vist at IA32 platformen er meget oktalt orienteret, hvilket synes at være oversat af alle, selv i Intels egen dokumentation. Dette er væsentligt at forstå hvis man vil implementere en assembler, og det letter en række opgaver meget. Med dette dokument i hånden bliver det forholdsvist simpelt at implementere de forskellige instruktioner.

Jeg har konstrueret et klassehierarki til repræsentation af en mindre mængde af IA32 instruktionssættet. Først er der en base klasse, `instruction`, der eksponerer to metoder: `get_encoded` og `get_size`. `get_size` returnerer antallet af bytes som en instruktion kræver. Denne størrelse benyttes af assembleren til at afgøre antallet af bytes der skal springes når det er tilfældet. `get_encoded` returnerer en `vector` af `char`'s, det vil sige en samling bytes der svarer til instruktionens binære repræsentation. Det er disse der skal konkateneres sammen for at få det endelige program.

Jeg konstruerer en anden base type, `operand`, der i sig selv er abstrakt men som specialiseres i de typer af operander der findes: immediates, dvs. konstanter der er direkte "spundet" ind i koden, registre samt variable i hukommelsen. Nu er instruktionssættet desværre ikke så fleksibelt at alle instruktioner accepterer fuldstændigt arbitrære operander - der er mange begrænsninger og specialtilfælde. Derfor kan man ikke bare lade en instruktion acceptere et antal operander. Hvis vi kigger på en central instruktion som `Mul`, der udfører en heltals gangeoperation, kan vi se at der ifølge Intels dokumentation findes disse varianter, hvis vi ser bort fra 64-bit udvidelser:

- `Mul AL, r/m8`
- `Mul AX, r/m16`
- `Mul EAX, r/m32`

(`r/mxx` betyder et register eller en variabel (hukommelsesposition) af `xx` bits). Semantikken er her, at man kan gange accumulator registret med et register eller en variabel. Resultatet lagres i henholdsvis `AX`, `DX:AX` og `EDX:EAX`, dvs. 8-bit udgaven gemmer resultatet i sit tilsvarende 16-bit register, mens 16 og 32-bit udgaverne benytter accumulator og data (`DX`) registret for at kunne holde den samlede værdi. Dette viser lidt om instruktionernes særlige egenskaber. Selv om `Mul` blot ganger to operander

<sup>4</sup> Googles cachede udgave [<http://groups.google.dk/groups?selm=3gmn58%24rbk%40DGS.dgsys.com>]. Dokumentet ligger desuden sammen med kildekoden til projektet her, indlejret i `RTCG.cbi` filen.

sammen, skal den første operand i dette tilfælde være en udgave af accumulator registret, og man skal være opmærksom på at resultatet ikke alene placeres heri men også muligvis i data registret som derfor ikke overlever en `Mul` instruktion. Endelig ser vi at selv om den anden operand kan variere er ikke alle typer af operander understøttede - man kan ikke bruge immediates.

C++ har ikke et "pattern matching" koncept som for eksempel ML, men man kan opnå noget der kommer derhen ad ved hjælp af simpel polymorfi. Jeg laver en type for hvert register, alle navngivet med præfikset `reg`, og som alle arver fra klassen `reg`, der igen arver fra `operand`. Da disse refererer til registre der findes i cpu'en og derfor altid er til stede, laver jeg en globale instansieringer af dem alle, for at lette kodeskrivningen.

```
const regAL AL;
const regAX AX;
const regEAX EAX;
const regCL CL;
const regCX CX;
// ...
```

Det giver mulighed for at man kan lave en funktion der accepterer en operand, et hvilket som helst register eller et specifikt register som parameter. Hvis `Mul` eksempelvis skulle implementeres i form af en funktion af en art, ville man kunne lave følgende deklarationer, forudsat at vi har operandtyper for variable af forskellig størrelse defineret som `v8`, `v16` og `v32`:

```
void Mul(regAL, r8 reg);
void Mul(regAX, r16 reg);
void Mul(regEAX, r32 reg);
void Mul(regAL, v8 var);
void Mul(regAX, v16 var);
void Mul(regEAX, v32 var);
```

Disse funktioner accepterer kun accumulatorregistre som første parameter og et vilkårligt register eller en variabel som anden parameter og hvis man skulle forsøge at kalde dem med noget andet vil oversætteren melde fejl, så man opdager det tidligt. (Desuden vil man kunne få hjælp fra moderne IDE'er, der kan hjælpe med at vise lovlige parametre mens man skriver).

Således konstruerer jeg udgaver af de instruktioner som jeg har brug for i denne første udgave af systemet. En anden klasse, `assembler` har ansvaret for at holde styr på en samling af disse instruktioner og for at kunne kalde det samlede program. De eneste instruktioner der kræver en anden uddybelse er `Jump` og `Call`. Disse er specielle fordi spring i koden skal angives relativt - man springer et antal bytes frem eller tilbage i koden. Derfor bliver man nødt til at kende både den gældende instruktions placering i hukommelsen samt det sted, man ønsker at springe til. Man kunne "snyde" og lave en `Jump`, der blot initialiseres med et offset, og så kræve at programmøren selv holder styr på hvor langt der skal springes. Det ville være tæt på ubrugeligt og meget svært at vedligeholde. I stedet lader jeg `assembler` klassen vedligeholde et system af labels, som man kan registrere undervejs. Når man har færdiggjort sin kode, løber assembleren alle instruktioner igennem og beregner offsets fra `Jump` instruktioner til de givne labels. Ligeledes fortæller den `Call` instruktioner om deres absolutte placering i hukommelsen, så de kan beregne deres offset til den funktion man ønsker at kalde.

## 2.3. Dynamiske konstruktioner

### 2.3.1. Opbygning og syntaks

Selv om det at kunne assemble kode er et skridt i den rigtige retning kan vi godt gå videre, så man ikke behøver at memorisere hele IA32 instruktionssættet selv om man vil benytte denne strategi. Ved at skabe et nyt abstraktionsniveau oven på assemblerniveauet, kan vi lave et mere C-lignende sprog. Selv om man kan indvende at C er et meget simpelt sprog der næsten oversættes direkte til assembler, men erfaringen har vist at C skalerer endog meget stabilt - og det udgør grundkernen i de fleste operativsystemer og meget anden kode rundt omkring. Hvis vi kan lade brugeren skrive dynamisk kode i et

sprog der ligner C vil vi rent faktisk have et framework der vil kunne bruges til store projekter.

Og hvordan kan vi så gøre det? Endnu en gang har vi muligheden for at parse strenge hvilket vil give os mulighed for at opnå præcist den syntaks vi gerne vil. Og endnu en gang præsenterer det samme problem sig, nu i endnu højere grad, nemlig at dette så først vil blive leksikalskt og semantisk valideret på kørselstidspunktet. Igen kan vi afhjælpe dette med C++ typer. Vi indfører et antal typer og funktioner:

- `int_d`
- `bool_d`
- `id_f`
- `return_d`

Alle disse `_d`'er er dynamiske pendanter til de statiske varianter i almindelig C(++). Man kunne udvide listen, men disse elementer burde være nok til at illustrere den fremgangsmåde jeg har i tankerne. `int_d` og `bool_d` er typer som man kan instansiere. De er forskellige fra de statiske ved, at de, når de instansieres (statisk) genererer kode der initialiserer dem, præcist som en oversætter gør på det sted hvor de initialiseres. Herudover benytter jeg C++'s mulighed for at overskrive operatører, således at man kan behandle dem som almindelige typer der kan lægges sammen, sammenlignes og så videre, men hvor disse operationer konstruerer kode snarere end at fore tage en egentlig sammenligning.

Det er meget fint med `int_d` og `bool_d` - de er forholdsvis enkle at implementere. Lidt mere besværligt bliver det med flow manipulation som løkker og betingede spring. Man kan ikke "overskrive" de indbyggede konstruktioner som `if` og `while`, der tjener disse formål, på samme måde som man kan overskrive operatører. Og godt det samme, for vi har stadig brug for de almindelige, statiske udgaver. Hvad gør vi så? Man kunne løse det helt trivielt ved at stille nogle mere simple elementer til rådighed som `goto_d` og måske en `goto_if_d` eller lignende, men selv om C godt nok understøtter `goto`, har denne konstruktion været upopulær siden Dijkstra pointerede at den let medfører dårlig kode. Den løsning jeg har lavet gør at man kan bruge en syntaks der nærmer sig almindelig C syntaks. Den er lidt anderledes - nok så anderledes at man lige skal vænne sig til den, men dog ikke så anderledes at man vil være et øjeblik i tvivl om hvad er stump kode gør. Betragt følgende eksempel:

```
if_d(x == 0);
{
    scope s(this);
    y = 100;
}
else_d();
{
    scope s(this);
    y = 200;
}
```

Skrevet som statisk kode ville dette se således ud:

```
if(x == 0)
{
    y = 100;
}
else
{
    y = 200;
}
```

Lad os se på forskellen. Den første forskel er semikolon'et efter `if`-sætningen. Det optræder ikke i den statiske version, da sætningen ikke betragtes som afsluttet endnu. Semikolon'et er nødvendigt fordi

`if_d` i virkeligheden er et funktionskald. Dette funktionskald konstruerer kode der, givet en `bool_d`, enten springer over eller udfører den kommende kode. Og hvordan kan den så vide hvad "den kommende kode" er? Det er lidt besværligt. Man kunne vælge at sige, at det er den næste linie dynamisk kode, men hvordan kan man vide hvor en sådan linie ender? Den kan være sammensat af en række operationer, hvoraf ingen ved noget om den leksikalske struktur. Jeg har valgt i stedet at pålægge programmøren en mindre byrde - nemlig at skulle instansiere et `scope` objekt, som `s` variabelen, den anden forskel på de to stykker kode. Et `scope` er en indikator for det, der går under samme navn i C - nemlig en afgrænsning af en `{}` og en `}`. Inde i et `scope` er variable lokale, og operationerne i dette `scope` er ligeledes lokale - til vores `if` sætning. `scope` klassen benytter et C++ idiom kaldet RAII, Resource Aquisition is Initialization. Det vil sige at klassen benytter C++'s automatiske kald af destruktører på lokale objekter ved udgangen af et `scope`. Når det omsvøbende `scope` lukkes, destrueres `s`, og under denne destruktion ligger logik der registrerer en label som `if`-sætningen kan springe til i tilfælde af at udtrykket var falsk. Endelig gælder det for `else_d` som for `if_d`, at den i virkeligheden er et funktionskald og derfor skal have `()`; efter.

## 2.3.2. Implementation

De dynamiske variable er bedst illustreret ved at se på en delmængde af koden til `int_d`:

```
class int_d : public variable_d<int>
{
public:
    // ...

    int_d& operator += (i32 const& i)
    {
        assembler* a = g_environment.get_assembler();
        a->add_instruction(new Mov(EAX, get_ptr()));
        a->add_instruction(new Add(EAX, i.value));
        a->add_instruction(new Mov(get_ptr(), EAX));
        return *this;
    }

    // ...

    bool_d operator == (int_d& v)
    {
        assembler* a = g_environment.get_assembler();
        bool_d temporary;

        a->add_instruction(new Mov(EAX, get_ptr()));
        a->add_instruction(new Mov(EBX, v.get_ptr()));
        a->add_instruction(new Cmp(EAX, EBX));
        a->add_instruction(new Mov(EAX, imm<32>(0)));

        // Set AL (and hence EAX) to 1 if EAX and EBX
        // were equal.
        a->add_instruction(new Set(AL, EQUAL));

        a->add_instruction(new Mov(temporary.get_ptr(), EAX));
        return temporary;
    }

    // ...
};
```

En `int_d` arver fra `variable_d`, der indeholder en instans af den egentlige type, dvs. en `int_d` indeholder en `int` som den kan tilgå gennem funktionen `get_ptr`, der returnerer en pointer til den. Denne `int` er allokeret på den såkaldte free store. Senere kunne det måske være ønskeligt at allokere plads til disse variable på en stak lokalt i den genererede funktion, for at vinde lidt ydelse. `g_environment` er det miljø som koden skrives i. Det kommer jeg nærmere ind på senere. Det in-

teressante på dette tidspunkt er at se på de to operatoren. Den første er den velkendte akkumuleringsoperator, +=. Som det ses, foretages der ingen addition her, men derimod konstrueres en lille stump kode. Først hentes værdien af variabelen ind i registret EAX. Derefter lægges konstanten, som denne udgave af operatoren modtager, til EAX og endelig flyttes den nye værdi af EAX tilbage i variabelen. Det, der returneres er `this` pointeren, det vil sige variabelen selv. Dette er C standard og selv om det kan synes meningsløst i dette tilfælde da variabelen ikke manipuleres af selve kaldet, er det det ikke da det betyder at man kan på denne måde kan sammensætte operatoren på en måde der er konsistent med C, hvor det eksempelvis giver mening at skrive følgende:

```
int x = y += 2;
```

Her inkrementeres `y` først med to hvorefter `x` tildeles den nye værdi.

Den anden operator, lighedstestoperatoren, er interessant fordi det, den returnerer er en `bool_d` - altså en dynamisk bool. Det er nødvendigt fordi lige som værdien af en dynamisk integer er ukendt ikke bare på oversættelsestidspunktet men også på det tidspunkt hvor lighedstestoperatoren her kaldes, bliver vi nødt til at oprette en midlertidig, dynamisk boolsk variabel til dette formål. Vi henter værdierne af de to variable ind i henholdsvis EAX og EBX og laver en sammenligning. Resultatet placeres i EAX, og den midlertidige variabel tildeles værdien af EAX.

For at have et samlingspunkt for alt dette laver jeg en klassetemplate kaldet `functor`. Denne arver fra `function` templateen der er at finde i `Boost::function` og sandsynligvis den kommende standard. Dette forærer mig en avanceret implementation af anvendelsesoperatoren der virker med alle funktionssignaturer. Man konstruerer en `functor` med en instans af en `functor_base`. Denne skal indeholde selve den kode man vil lave.

Med hensyn til de dynamiske if-sætninger bliver teknikken lidt mere kompleks. Den tidligere omtalte variabel `g_environment` er som sagt det "miljø" som koden bliver indsat i. Denne stiller en assembler instans til rådighed, som man kan indsætte sine instruktioner i. Derudover vedligeholder den en stak af `scope_proxy`'er, der er en abstrakt klasse som specialiseres i form af klassen `scope`, der findes for en enkelt udgave af `functor_base`. Når man i sin dynamiske kode instansierer et `scope`, lægges det oven på denne stak, og når det opløses igen registrerer det nogle labels osv. Lad os se nærmere på `pop_scope` i `g_environment`:

```
void pop_scope()
{
    scope_proxy* oldscope = scopes_.top();
    scopes_.pop();

    if(oldscope->get_scope_type() == scope_proxy::IF)
    {
        // If this was an "if" scope, we'll begin by
        // jumping to the "endif" label, and then register
        // both the endif and the else label.

        std::string else_label = oldscope->get_label_1();
        current_endif_label_ = oldscope->get_label_2();

        get_assembler()->
            add_instruction(new Jmp(current_endif_label_));

        get_assembler()->register_label(current_endif_label_);
        get_assembler()->register_label(else_label);
    }
    else if(oldscope->get_scope_type() == scope_proxy::ELSE)
    {
        // It was an "else" scope. Move the "endif" label down
        // to the end of this scope then.

        current_endif_label_ = oldscope->get_label_1();

        get_assembler()->register_label(current_endif_label_);
    }
}
```

```
}  
}
```

Vi kan se at der, når man lukker et scope der efterfølger en `if_d`, registreres to labels: "else" og "endif". De registreres det samme sted, nemlig efter koden i scope'et. Desuden indsættes en `Jump` til "endif", hvilket kan virke (og er!) unødvendigt, da denne label følger umiddelbart efter. Grunden er, at denne label gen-registreres hvis man efterfølgende laver et "else" scope. Denne funktionalitet kunne forsimples ved at lægge et abstraktionsniveau ind imellem det C-lignende sprog og assemblerniveauet, som man gennemløber før funktionen kan kaldes, hvor nødvendige labels og spring indsættes. En sådan udvidelse er meget oplagt af flere årsager - eksempelvis til en fremtidig allokering af registre, så det vil sandsynligvis være en del af en kommende udgave. Ikke desto mindre fungerer koden som den ser ud nu med blot ét gennemløb (hvilket i øvrigt betyder at selve kodegenereringen er hurtig, men det er nu ikke noget specielt interessant mål).

---

# Chapter 3. I praksis..

Jeg indledte opgaven med at forsvare at der kan være tilfælde hvor forøget kompleksitet, som et system som dette unægteligt medfører, er acceptabelt på grund af vundet ydelse. Programmeringssprogsdesignere klantræs ofte for at skabe "løsninger, der leder efter et problem", og for igen og igen at illustrere paradigmer ved hjælp af implementation af fakultetsfunktioner eller løsninger på dining filosofers problemet. Har dette framework en praktisk anvendelse og i så fald, hvori består den? Det vil jeg mene. Lidt simplificeret sat op kan man sige at en cpu der kører med klokfrekvens på 1GHz vil kunne afvikle 1 gigabyte kode på - om ikke et sekund så i hvert fald få sekunder. 1Gb svarer nogenlunde til hvad der befinder sig i skrivende stunds almindelige PC, og alligevel har de fleste deres PC tændt i mere en et par sekunder ad gangen. Grunden hertil er naturligvis at den kode der afvikles på deres maskiner er fyldt med løkker, der holder cpu'en fanget i en given mængde kode i lang tid. I næsten alle programmer findes et "mainloop", der afvikles igen og igen, mens de resterende måske 80% af programmet kun besøges sjældent. Hvis man ønsker at optimere et program kan det altså betale sig at kigge på den mængde kode der afvikles i mainloop'et.

## 3.1. Message Dispatching

Et design mønster der optræder ganske mange steder er message dispatching. Eksempelvis skal programmer der kører under Microsoft Windows benytte message dispatching for at kommunikere med operativ systemet. Mange kritiserer systemet og der er mange gode argumenter imod at benytte message dispatching som jeg ikke vil komme ind på her, men sikkert er det, at i hvert fald Windows programmer er tvungne til at benytte det i nærmeste (og sandsynligvis også den fjerne) fremtid. Mange abstraktionslag er blevet opfundet, der lægger sig oven på dette system, så programmøren ikke behøver at tænke på det men det ændrer ikke på at det er fundamentet nedenunder alligevel. Ved message dispatching forstås en art forsimplet, generaliseret funktionskald internt i eller på tværs af tråde og processer. Man sender en besked eller message, med et antal foruddefinerede parametre, og modtageren informeres om at en besked med en given identifikation er modtaget og kan herefter reagere derpå.

Måden det fungerer på er ofte at klienten bliver bedt om at konstruere en message handler, en funktion der er ansvarlig for håndtering af indkomne beskeder. Den har måske en signatur der ser således ud:

```
void HandleMessage(int message_number, int parameter);
```

Denne funktion kaldes så af den beskedsendende enhed (for eksempel operativsystemet), med beskednummer og tilhørende parameter. Hertil hører en kontrakt om hvilke beskeder der har hvilke numre, og hvad deres parameter skal indeholde. Alle disse situationer skal handleren så håndtere. En almindelig måde at konstruere en sådan handler ser således ud:

```
void MessageHandler(int message, int parameter)
{
    switch(message)
    {
        case OPENWINDOW:

            // Åben et vindue...

            break;

        case QUIT:
            exit();
    }
}
```

Her er OPENWINDOW og QUIT foruddefinerede konstanter som operativsystemet og klientprogram-

met er enige om. Problemet med denne fremgangsmåde er at den samler forskellig funktionalitet i én funktion. OPENWINDOW og QUIT har intet andet med hinanden at gøre end at de modtages som beskeder. For at skille det lidt ad, kan man lave det lidt om, så man kan "registrere" en funktion, der håndterer en given besked:

```
typedef function<void (int)> Handler;
map<int, Handler> > message_handler_map;

void MessageHandler(int message, int parameter)
{
    if(message_handler_map.count(message) != 0)
    {
        message_handler_map[message](parameter);
    }
}

void RegisterHandler(int message, function<void (int)> handler)
{
    message_handler_map[message] = handler;
}
```

Denne (let forsimplede) løsning er meget pænere - man kan registrere en arbitrær funktion til at håndtere en given besked. Dermed er det også blevet lettere at genbruge funktionalitet i forskellige handlede, at bruge objekt orienterede design mønstre med arv og så videre. Perfekt. Den eneste lille hage er, at vi nu har indført et ekstra funktionskald. Ikke bare dét - vi har indført et kald til en arbitrær funktion hvilket vil sige at det er umuligt for oversætteren at inline kaldet, eller blot optimere ud fra kendskab til funktionens placering i hukommelsen i forhold til handleren. Desuden har vi tilføjet et opslag i en map struktur. Det betyder måske ikke så meget, men eftersom en sådan message handler udgør den mest centrale del af mainloop'et i eksempelvis alverdens Windows programmer, vil man kunne spare en del ved at holde det så hurtigt og småt som muligt. Det er ikke ualmindeligt at et program modtager tusindvis af beskeder i sekundet. Her kan vi med fordel benytte vores framework. Hvis man undlader at skrive MessageHandler funktionen som statisk kode men derimod genererer den når vi ved hvilke beskeder vi vil håndtere, kan vi lave en specifik funktion der er optimeret til netop dette. Vi bliver nødt til at udvide koden lidt:

```
typedef function<void (int)> Handler;
map<int, Handler> > message_handler_map;
functor<void (int, int)>* MessageHandler

void RegisterHandler(int message, Handler handler)
{
    message_handler_map[message] = handler;
    GenerateMessageHandler();
}

void GenerateMessageHandler()
{
    struct NewHandler : public functor_base<void (int, int)>
    {
        NewHandler()
        {
            scope s1(this);

            for(map<int, Handler> >::iterator i =
                message_handler_map.begin();
                i != message_handler_map.end(); ++i)
            {
                int message = i.first;
                Handler handler = i.second;
            }
        }
    };
}
```

```

        if_d(*(param_[0]) == message);
        {
            scope s2(this);
            call_d(handler, *(param_[1]));
        }
    }
} NewHandlerInstance;

delete MessageHandler;
MessageHandler = new functor<void (int, int)>
    (NewHandlerInstance);
}

```

Nu er MessageHandler ikke en funktion men en functor, som bliver genereret hver gang der registreres en ny message. Dette er dynamisk også i den forstand at man kan registrere midlertidige messages under kørslen osv. Denne konstruktion er, efter min mening, den bedste måde at opnå optimal ydelse samtidig med at man bevarer muligheden for at holde koden adskilt <sup>5</sup>.

## 3.2. Bitblt

En bitblt, eller bit block transfer, rutine er termen for en funktion der flytter en blok af data fra ét sted i hukommelsen til et andet. Nærmere betegnet er disse data bitmap data - grafik. Det lyder trivielt of det er det for så vidt også. Den helt simple udgave kopierer simpelthen som et almindeligt kald til memcpy. Det bliver dog lidt mere kompliceret når man for eksempel ønsker at rutinen skal udføre clipping. Ofte vil man nøjes med at kopiere et udsnit af sine data, og derfor bliver man nødt til at indføre at kopieringen kan springe dele over for hver linie, eksempelvis. Af og til ønsker man også at lave en konvertering undervejs - hvis data det ene sted ligger som 16 bit per pixel og som 24 bit per pixel det andet. I så fald involveres et antal logiske operationer og shifts for hver pixel. Af og til ønsker man at blande data'ene snarere end blot at overskrive dem. Hvis man laver alpha blanding, skal destinationsdataene udgøre en blanding af det, de var tidligere og det, man kopiere ind oven i. Endelig hænder det at man ønsker at lave geometriske transformationer, dvs. ikke blot kopiere ind i samme størrelse men måske skalere eller rotere sine data.

Da et bitblt kald ofte involverer data der kan fylde en hel computerskærm hvilket for eksempel kan være 1024\*768 pixels af 32 bit, dvs. 3145728 bytes, og man ofte forventer sin grafik opdateret for hver gang skærmen blinker, dvs. 70 gange i sekundet, bliver det tydeligt at man ikke kan sløse alt for meget med sin implementation. Det er klart at det er hurtigst at lave den helt simple kopiering, hvis det er muligt. Udgaven der både klipper, konverterer, blander og transformerer er klart langsommere end en simpel kopiering. Vi vil gerne undgå så meget som muligt. Da kopieringsløkken gennemløbes mange gange ses det at det er suboptimalt at lave betingede spring for hver mulig tilføjelse til operationen - og det vil være irriterende fordi hvert spring vil være det samme for hver enkelt pixel. Desværre er der alt for mange kombinationsmuligheder til at man kan skrive specialiserede udgaver af dem alle. I dette tilfælde er det oplagt at konstruere en optimal funktion når man ved hvilke funktionaliteter man har brug for.

## 3.3. STL functional

Et andet interessant anvendelsesområde kunne være at finde i STL's funktionsbibliotek som nævnt tidligere. Hvis man laver specialiserede udgaver af bind vil man måske kunne anvende STL kode der bag facaden laver dynamisk genereret kode. Dette har jeg endnu ikke udforsket, så det må blive et senere

<sup>5</sup> Det skal dog nævnes at denne løsningen her er teoretisk bedre men at den med den nuværende udgave af framework'et ikke vil være meget bedre i praksis da functor klassen selv foretager et antal ekstra funktionskald når den anvendes. Det betyder mindre i andre situationer hvor man vil pakke selve den kritiske løkke ind i en dynamisk funktion snarere end at kalde en dynamisk funktion fra en kritisk løkke.

projekt.

---

# Chapter 4. Afprøvning

Men virker det så? Det mest interessante ville naturligvis være at sammenligne to udgaver af en kompleks parameteriseret algoritme - en "almindelig" og en dynamisk genereret, og se hvad forskellen i ydelse er. Grundet den tidlige tilstand frameworket befinder sig i har jeg ikke lavet en så omfattende afprøvning. Desuden vil det være svært at vise nogen synderligt imponerende ydelse med den kode der genereres på nuværende tidspunkt, og en generator der laver rigtig optimeret kode er et stort arbejde der langt overskrider ambitionerne for dette projekt. Til gengæld er jeg overbevist om at de indre dele i frameworket fungerer. Dette hænger sammen med den måde jeg har udviklet koden - jeg har nemlig skrevet den "test driven", hvilket vil sige at jeg for hver ny komponent først har skrevet en såkaldt unit test, der afprøver den nye funktionalitet, og først derefter skrevet selve komponenten. Disse tests bliver ved med at eksistere og tjener som garant for at jeg ikke på et senere tidspunkt kommer til at lave en ændring eller tilføjelse der bevirker at gammel kode bliver ødelagt.

Assemblerdelen afprøver jeg ved at sammenligne genereret kode for hver enkelt instruktion med kode genereret af inline assembleren som C++ understøtter. Jeg laver nogle små makroer der hjælper mig med at "trække" den genererede kode ud givet to labels. Denne kode kan jeg så sammenligne med den kode der kommer ud af hver af mine instruktioner. En simpel test af i dette tilfælde Inc instruktionen ser således ud:

```
void instructions_test::TestInc()
{
    auto_ptr<Inc> i1(new Inc(AL));
    auto_ptr<Inc> i2(new Inc(BL));
    auto_ptr<Inc> i3(new Inc(EBX));
    auto_ptr<Inc> i4(new Inc(CX));

    __asm pushad
    a_1: __asm Inc AL
    a_2: __asm Inc BL
    a_3: __asm Inc EBX
    a_4: __asm Inc CX
    a_5: __asm popad

    DEFINEVECTOR(code1_real, a_1, a_2);
    std::vector<char> code1_synthesized = i1->get_encoded();
    CPPUNIT_ASSERT(code1_real == code1_synthesized);

    DEFINEVECTOR(code2_real, a_2, a_3);
    std::vector<char> code2_synthesized = i2->get_encoded();
    CPPUNIT_ASSERT(code2_real == code2_synthesized);

    DEFINEVECTOR(code3_real, a_3, a_4);
    std::vector<char> code3_synthesized = i3->get_encoded();
    CPPUNIT_ASSERT(code3_real == code3_synthesized);

    DEFINEVECTOR(code4_real, a_4, a_5);
    std::vector<char> code4_synthesized = i4->get_encoded();
    CPPUNIT_ASSERT(code4_real == code4_synthesized);
}
```

Det, der sker her er, at jeg først instansierer fire udgaver af Inc instruktionen - min udgave. Herefter laver jeg tilsvarende instruktioner med inline assembleren, og endelig afprøver jeg at hver genereret kodelinje svarer til det, inline assembleren genererer. Mit projekt er sat op så alle disse tests afvikles hver gang jeg bygger, hvilket gør at jeg bliver informeret meget hurtigt hvis en ændring jeg har foretaget har ødelagt noget et eller andet sted.

Det højere abstraktionslag afprøver jeg ved at skrive noget kode der skal returnere en bestemt værdi, der bliver efterprøvet. Dette tjener også som en grundigere afprøvning af assembleren, da denne er en

forudsætning for at de højere niveauer fungerer. Her er et eksempel på afprøvning af dynamisk if:

```
// ...
struct fct3 : public functor_base<int()>
{
    fct3()
    {
        int_d x(1);
        int_d y(2);

        if_d(x == y);
        {
            scope s1(this);
            return_d(100);
        }
        else_d();
        {
            scope s2(this);
            return_d(200);
        }
        return_d(300);
    }
} fct3_instance;

functor<int()> f3(fct3_instance);

CPPUNIT_ASSERT_EQUAL(200, f3());

// ...
```

Det lille program der genereres her bør returnere 200, hvilket jeg efterprøver med assert-sætningen til sidst. Programmet giver i øvrigt følgende maskinkode:

```
004C15E0  push     edi
004C15E1  push     esi
004C15E2  push     ebx
004C15E3  push     ebp
004C15E4  mov     ebx,esp
004C15E6  add     ebx,14h
004C15EC  mov     eax,1
004C15F1  mov     dword ptr ds:[004C08D8h],eax
004C15F6  mov     eax,2
004C15FB  mov     dword ptr ds:[004C0B90h],eax
004C1600  mov     eax,dword ptr ds:[004C08D8h]
004C1605  mov     ebx,dword ptr ds:[4C0B90h]
004C160C  cmp     eax,ebx
004C160E  mov     eax,0
004C1613  sete   al
004C1616  mov     dword ptr ds:[004C0BD0h],eax
004C161B  mov     eax,dword ptr ds:[004C0BD0h]
004C1620  mov     dword ptr ds:[0032B7E0h],eax
004C1625  mov     eax,dword ptr ds:[0032B7E0h]
004C162A  cmp     eax,0
004C162F  je     004C1644
004C1635  mov     eax,64h
004C163A  pop     ebp
004C163B  pop     ebx
004C163C  pop     esi
004C163D  pop     edi
004C163E  ret
004C163F  jmp     004C164E
```

```

004C1644 mov     eax, 0C8h
004C1649 pop     ebp
004C164A pop     ebx
004C164B pop     esi
004C164C pop     edi
004C164D ret
004C164E mov     eax, 12Ch
004C1653 pop     ebp
004C1654 pop     ebx
004C1655 pop     esi
004C1656 pop     edi
004C1657 ret

```

Dette er taget ud fra mit IDE's indbyggede disassembler, hvilket forklarer adresserne i siden, der refererer til hver enkelt instruktions absolutte placering i hukommelsen. Her illustreres det med al ønskelig tydelighed hvor uoptimeret den genererede kode er, men at den dog er semantisk korrekt.

De første 6 instruktioner er funktions entry-point kode, der lægger nogle kritiske registre på stakken, henter parametre ind osv.

Herefter, i instruktionerne 004C15EC til 004C15FB, tildeles de to variable (x og y) værdierne 1 og 2.

For at udføre sammenligningen `if_d(x == y)`, hentes de to variable ud i henholdsvis EAX og EBX i instruktionerne 004C1600 og 004C1605, og sammenlignes i instruktionerne 004C160C til 004C1613.

Nu indeholder EAX en dynamisk boolsk "temporary" - altså en variabel uden navn, som videreføres fra betingelsen til `if_d` funktionen. For at gøre dette gemmes den i hukommelsen i instruktionen 004C1616.

I de efterfølgende instruktioner, 004C161B til 004C1625 hentes den ud fra hukommelsen og ind i EAX, tilbage i hukommelsen på en ny adresse og atter ind i EAX. Grunden til at det sker to gange er, at C++ selv opretter en "temporary", der bliver kopieret til `if_d` sætningen. Dette bevirker at copy constructor'en kaldes, og der skabes en ekstra kopi. Alt dette er komplet unødvendigt da værdien allerede befinder sig i EAX registret, og dette ville for eksempel en peephole optimizer<sup>6</sup> let opdage og fjerne.

Nu (instruktion 004C162A) sammenlignes EAX, dvs. det boolske resultat af sammenligningen, med nul.

Hvis EAX var nul, springes fra 004C162F til 004C1644 - vores "else" kode. Hvis ikke, fortsætter koden i instruktion 004C1635 med at placere værdien 0x64 (100) i EAX, hvilket er den returnværdi vi ønsker at returnere med hvis betingelsen var sand.

De efterfølgende instruktioner 004C163A til 004C163E er exit-point kode, der løfter de gemte registre fra stakken og returnerer. Herefter kommer en spring instruktion, der hopper ned til "end if", det vil sige efter else-scopet. Denne instruktion er unødvendig da den kommer umiddelbart efter en "ret" instruktion, men vi ikke havde returneret inde fra if-scopet, ville dette sikre at vi kun afvikler enten if- eller else-koden.

Else koden, der går fra 004C1644 til 004C164D gør stort set det samme, med den forskel at EAX sættes til 0x8C (200). Endelig kommer til sidst den kode, der returnerer efter if-sætningen, der sætter EAX til 0x12C (300) og returnerer. Da det gælder at 1 ikke er lig med 2 burde funktionen returnere 200, hvilket også er det, jeg efterprøver i assert-sætningen i testen.

<sup>6</sup> En peephole optimizer er et lille program der kigger på en meget isoleret del af koden ad gangen og ser efter om to instruktioner gør hinanden redundante - og fjerner i så fald den ene.

---

# Chapter 5. Fremtidige milepæle og andre systemer

Projektet her er nok færdigt som en første udgave, men der er mange ting der skal forbedres før det kan tages i brug i praksis. Her er en list af ting jeg gerne vil have rettet eller tilføjet:

- Platformsafhængighed. Projektet er platformsafhængigt i ordets strengeste forstand. Ikke alene er det afhængigt af en bestemt familie af processorer, det kræver også et bestemt operativsystem og endda en speciel oversætter. Afhængigheden af en CPU er svært at ændre, da man stort set vil være nødt til at skrive det hele for hver CPU det skal understøtte. Men operativsystem og oversætter vil man nogenlunde let kunne abstrahere bort fra. Den eneste binding til operativsystemet består i et kald til `FlushInstructionCache`, der er Win32 specifik, og som sørger for at et givet område i hukommelsen, der netop er blevet ændret, ikke ligger i en gammel udgave i cpu'ens cache. Der er tilsvarende funktioner i andre operativsystemer, og en så simpel binding er let at abstrahere bort fra. Lidt vanskeligere er det med oversætteren. For det første kræver mine tests at inline assembleren genererer kode der ser ud som de udgaver af instruktionerne jeg har lavet. IA32 kode er ikke entydig i den forstand at der kan være flere måder at beskrive den samme instruktion på - med forskellige størrelser på operanderne etc. Endelig skal man være meget omhyggelig med at opfylde kaldkonventionerne der desværre varierer meget fra oversætter til oversætter. Dette er noget Bjarne Stroustrup har fortalt mig at han selv beklager, men at der eksisterer initiativer der forsøger at unificere dette på tværs af oversættere. Det findes dog ikke på dette tidspunkt, og forkert opførsel i forhold til kaldkonventioner kan resultere i meget mærkværdige fejl der er svære at opdage og endnu sværere at rette.
- `g_environment` er ikke tråd-sikker. Hvis man forsøger at skrive dynamisk kode fra multiple tråde på samme tid kan der opstå problemer fordi `g_environment` kun vedligeholder én udgave af assembleren.
- Den genererede kode er sub-optimal, for at sige det mildt. Det er klart at et framework der kun vil have anvendelse i tilfælde hvor ekstrem ydelse er påkrævet, bliver nødt til at skabe kode der yder bedre end tilfældet er lige nu. Rigtig register allokering og optimering der fjerner unødvendige instruktioner vil være påkrævet som minimum.
- Flere konstruktioner. Det vil være oplagt at udvide listen med `while_d`, `for_d` og så videre. Desuden kunne man måske arbejde med structs ud over blot primitive typer.
- Vektorisering. Denne udgave af assembleren understøtter ikke de nyere instruktionssæt som MMX, SSE osv., der er særdeles velegnede til vektoriseret kode. Det ville være meget nyttigt at have nogle konstruktioner til at skabe sådan kode, især til områder som signalbehandling etc.

Endelig bliver jeg nødt til at nævne at jeg ikke er pioner på dette område. Der findes mange andre initiativer af forskellig art. Termen multi-stage programming indkapsler hele denne type af kodeskrivning, og der findes sprog som MetaML og MetaOCaml og TickC, der understøtter det direkte. Det giver naturligvis en række fordele, især i det at man kan arbejde mere naturligt med den dynamiske del af koden. Til gengæld er sprogene små og deres popularitet marginal. Det giver en anden type af problemer eller i hvert fald besværligheder - nemlig at der findes meget få værktøjer der understøtter dem, der er større sandsynlighed for fejl i dem (givet at der ikke er mange der arbejder med dem), og det vil være svært at finde biblioteker der benytter den nye funktionalitet.

Peter Sestoft har undersøgt hvordan man kan lave noget tilsvarende i C# og Java. Da disse sprog er designet til at blive afviklet på virtuelle maskiner har det den fordel at man i så fald kan skrive "platformsafhængig" kode med tryghed i sindet - denne platform ligger oven på andre platforme og derved har man platformsuafhængigheden alligevel.

Et lignende projekt, GNU Lightning, stiller samme funktionalitet (på assemblerniveau) til rådighed for brugeren, i C. Dette projekt minder meget om mit og er tilmed skrevet til flere platforme - både x86, SPARC og PowerPC i følge hjemmesiden. Den væsentligste forskel fra mit projekt er, at syntaksen skal indkodes ved hjælp af makroer, det vil sige ved hjælp af C præprocessoren. Dette betyder sandsynligvis at der vil være svagere typecheck på oversættelsestidspunktet (C præprocessoren er typeløs), hvilket kan resultere i en større mængde fejl der først opdages ved afvikling.

---

# Chapter 6. Konklusion

Mit mål var at skabe et framework der gør at man kan lave kode, der bliver dynamisk genereret under afviklingen. Det mener jeg at have opnået, om end med et par begrænsninger. På nuværende tidspunkt er framework'et ikke meget værd i sig selv ud over som udgangspunkt for mere robust system der udfører reel optimering på den skabte kode. Designet viser at man kan lave konstruktioner i C++ som gør det muligt at skrive den slags kode med en nogenlunde rimelig syntaks hvilket jeg mener er det vigtigste, da det gør at man let vil kunne udbygge og videreudvikle den underliggende kode og være tryk ved at man stadig vil kunne bruge det til rigtig udvikling.

---

# Referencer

Veldhuizen, Todd. *C++ Templates as Partial Evaluation* 1999

<http://osl.iu.edu/~tveldhui/papers/pepm99.ps>

Hopkins, Mark. *A Summary of the 80486 Opcodes and Instructions* 1995

<http://groups.google.dk/groups?selm=3gmn58%24rbk%40DGS.dgsys.com>

Sestoft, Peter. *Runtime Code Generation with JVM and CLR* 2002

<http://www.dina.kvl.dk/~sestoft/rtc/rtcg.pdf>

*IA-32 Intel® Architecture Software Developer's Manual* 2004

<http://www.intel.com/design/Pentium4/documentation.htm>

Dijkstra, Edsger W.. *Go To Statement Considered Harmful* 1968

<http://groups.google.dk/groups?selm=3gmn58%24rbk%40DGS.dgsys.com>

---

# Appendix A. Kildekode