Universitetet i Oslo
Institutt for informatikk

Kompilering av mindre Python-moduler til C++

Robert Bauck Hamar

Masteroppgave

1. august 2007

Kompilering av mindre Python-moduler til C++

Robert Bauck Hamar

1. august 2007

Abstract

In order to achieve shorter execution times of Python code, I look at means of compiling Python into C++. To perform this task, I have written a Python compiler, and this compiler generates C++ code from a subset of the Python language.

Code generated from this code can in some cases expect to run in about 1/100 of the time used by ordinary Python.

I will look at the principles behind this compiler, and look at its (quite readable) generated code. Off the form of this generated code, I will explain why the compiler emits this fast code.

The compiler is written to be extended by the user, and this thesis also functions as a manual for the system.

The rest of this thesis is written in Norwegian Riksmål.

Innhold

1 Innledning

1.1 Bakgrunn

For lettere å kunne programmere en datamaskin, har menneskeheten skapt mange programmeringsspråk. Grunnen til at man ikke kan holde seg til ett, er at de forskjellige språkene har forskjellige egenskaper, noe som gjør at språkene egner seg til forskjellige oppgaver – selv om man gjennom Church-Turing-tesen har vist at det ikke er noen begrensning i hvilke oppgaver man kan utføre med noen av språkene. Det er særlig to språk jeg kommer til å gå inn på, og det er C++ og Python.

1.1.1 C++

C++ er definert av en internasjonal standard (se [4]), utgitt av standardorganet ISO. Det er så opp til kompilatorimplementører å lage en kompilator for språket. C++ har heller ikke andre krav til maskin og operativsystem enn C har, noe som gjør språket veldig portabelt. Ulempen med dette er at standarden da ikke inkluderer egenskaper mange trenger, slik at man blir låst til plattformer på grunn av tredjepartsbiblioteker. Standarden definerer for eksempel intet godt interaktivt brukermiljø, fordi enkelte datamaskiner ikke vil kunne støtte dette. Skal man da ta i bruk noe grafisk brukergrensesnitt i C++-programmer, må man da benytte biblioteker som Windows API, Xlib, Gtk og Qt. Verden ellers er dog heldigvis kommet godt i gang med å lage biblioteker som kan brukes på flere plattformer.

Størrelsen på C++-standarden er helt klart C++' store ulempe. Mange synes det er vanskelig å lære C++ (uten at jeg skal dokumentere dette), og det tar ofte tid å sette seg inn i språket. Få kompilatorleverandører har klart å implementere hele språket, og de fleste av dem synes å ha slått seg til ro med at nøkkelordet *export* ikke er implementert[9] og [10].

C++ er et programmeringsspråk med støtte for flere teknikker ([8, Kapittel 2]). Disse inkluderer da «modulær programmering» (støtte for *namespace* og flere kompileringsenheter), «dataabstraksjon» (støtte for å lage egne konkrete typer), «objektorientert programmering», «generisk programmering» (*template*) og «prosedural programmering» (funksjoner). Disse egenskapene gjør C++ velegnet til flere ting, og det blir brukt til mange ting. I tillegg passer Cs utbredelse C++ veldig godt, ettersom samhandling med C er spesifisert i C++-standarden. Dette – og selvfølgelig det at om ikke alle, så så godt som alle C++-kompilatorer kan kompilere C – gjør at C++ kan benytte seg av den store mengden C-biblioteker.

C++' egenskaper gjør dermed språket velegnet til flere typer prosjekter.

- Store prosjekter nyter godt av støtten for modulær programmering, som gjør det

enkelt å dele oppgaven mellom flere personer uten å risikere navnekollisjoner, og objektorientert programmering.

- Tidskritiske oppgaver nyter godt av nærhet med maskinvare samt støtten for generisk programmering, som også kan benyttes til å løse problemet (arvet fra C) med aliasing og pipelining. C er ansett som et meget effektivt språk, men klarer ikke å slå fortran, men [12, Blitz++] har klart å få C++ til å matche fortran. Ved hjelp av dataabstraksjon, kan dette også gjøres estetisk.

- Programmer som skal virke i et plassbegrenset miljø, nyter også godt av C++' egenskaper og filosofien om at man ikke betaler for noe man ikke bruker.

1.1.2 Python

Python er, i motsetning til C++, ikke spesifisert av en internasjonal standard, men alt som trengs for å lage en implementasjon, ligger fritt tilgjengelig på nettet; denne informasjonen kan lastes ned fra *http://www.python.org/*. Python er laget av Guido van Rossum, som vel har fått drahjelp senere. På hjemmesiden finnes også en ferdig Python-implementasjon, som er klassifisert som «open source», og kan lastes ned helt gratis. Så lenge man har sikret seg en kopi av dokumentasjonen, er dette nesten like godt som et standardisert dokument.

Pythons egenskaper ligner i rene syntaktiske elementer i stor grad C++'. Til sammenligning med C++, er likevel datamodellen kanskje den største forskjellen; Python har et dynamisk typesystem, og typene følger objektene, ikke navnene. Dette vil i mange tilfeller gjøre det umulig å dedusere en variabels type under kompilering. For å gjøre det mer formelt, kan man si: Python støtter objektorientering med multippel arv, akkurat som C++ gjør. Dynamisk binding sørger for å muliggjøre generisk programmering, dog uten *template*. C++ og python deler også støtte for såkalte «exceptions» – unntak er kanskje et godt nok norsk ord. Det eneste jeg savner ved Python er automatiske destructor-kall.

Til forskjell fra C++, er Python et skriptespråk. Skriptespråk virker typisk ved at en tolk (interpreter) tolker språket mens det kjører. Intuitivt sett, høres det da ut som om programmene vil kjøre tregere, og det vil de, men ettersom dagens datamaskiner er såpass effektive at programmer bruker mest tid på å vente på ordre fra brukeren, er det egentlig få programmer som trenger noe mer, og er det det, er det stort sett bare deler av programmet som går for tregt.

I tillegg tilbyr Python et grensesnitt mot C, så om det skulle være nødvendig, kan kritiske rutiner skrives i mere effektive språk. Mer objektivt sett, har skriptespråk ofte egenskaper vanlige kompilerte språk ikke har. Ettersom tolken må kjøres samtidig med programmet, er det ikke spesielt vanskelig å tillate kode fra for eksempel bruker, å evalueres, for eksempel for på en enkel måte å bygge inn en avansert kalkulator. En annen karakteristikk ved skriptespråk er at symboltabellene kan leses og endres under kjøring. Python er intet unntak – det tar i bruk nesten alle muligheter skriptemiljøet tillater.

1.1.3 C og Pythons grensesnitt mot C

C er et umåtelig populært språk, og mye av dette må vel tilskrives at enhver UNIX-installasjon kommer med en kompilator for dette språket. C ble opprinnelig skrevet for å skrive operativsystemet UNIX i, og har også nådd popularitet blandt andre operativsystemprogrammerere. C har etterhvert fått to standarder, C89 og C99, men det er fortsatt C89 som brukes mest.

Mange hevder at C er en delmengde av C++. Dette er ikke helt sant. Riktignok var det slik en gang på åttitallet, men etter det har språkene utviklet seg i forskjellige retninger. Bjarne Stroustrup, som skapte C++, var – og er fortsatt – veldig opptatt av typesikkerhet. Han utviklet konseptet *const*, samt spesifikasjon av argumenttyper ved funksjonsdeklarasjoner. C-verdenen så at dette hadde noe for seg, og lånte tilbake til C, dog med endringer, som ikke er direkte kompatibelt med C++. Men tar man C++ $\cap$ C, har man likevel et relativt bredt spekter. For eksempel tar scriptespråket Lua (*http://www.lua.org/*) sikte på å bare benytte denne mengden.

Som mye annet, er den vanlige Python-tolken skrevet i C, og et grensesnitt for alle som vil benytte seg av tolkerens muligheter i C-programmer, eller som vil utvide tolkerens muligheter med C-kode, er definert. Ettersom Python er et rikere språk enn C, er dette grensesnittet begrenset av C. C++ kan, som sagt, uten videre ta i bruk C-grensesnitter, men C++ har et rikere sett med egenskaper, som kunne være ønskelig å bruke i kommunikasjon med Python. Jeg vil diskutere disse egenskapene i de neste avsnittene.

Objektorientering

Objektivt sett, er det overraskende at støtten for objektorientert programmering alltid blir trukket frem. C++har fullstendig støtte for objektorientert programmering, men det tvinger på ingen måte programmereren å bruke det. Med fullstendig støtte, mener jeg:

Klasser En klasse er en sammensetning av objekter og operasjoner på disse.

Arv Arv lar en klasse få egenskaper fra en eller flere superklasser. Dette kalles også spesialisering, og definerer en «er en»-relasjon mellom klassene.

Multippel arv Støtte for at en klasse kan arve fra flere superklasser. Dette bør også medføre en mulighet for å unngå å arve de samme egenskapene flere ganger dersom superklassene arver fra samme klasse. Det siste er i C++ kalt virtuell arv. Multippel arv er en vanskelig egenskap, og er for eksempel ikke støttet av Java; Python støtter dog multippel arv.

Polymorfi Dette kalles også dynamisk, eventuelt sen, binding. Dette tillater klasser å overstyre de nedarvede operasjonene.[1]

Innkapsling Data kan innkapsles ved å begrense adgangen til dem, noe som hindrer andre deler av programmet å korruptere data.

[1] C++ støtter også andre former for polymorfi gjennom templates.

Python har begrenset støtte for objektorientering, idet språket ikke har skikkelig støtte for innkapsling. Nå har man en konvensjon om at navn som begynner med en understrek, ikke skal brukes utenfor klassen eller modulen, men jeg har opplevd å prøve å få nedlastede programmer, hvis skaper ikke har overholdt denne regelen, til å virke etter at senere versjoner av biblioteket endret sitt interne grensesnitt. Slikt skjer når man skriver programmer uten å kjenne språket godt nok. Mangelen på adgangsbegrensning har ingen innvirkning på språkets muligheter, men det reduserer muligheten for mennesklige feil.

C har ingen støtte for objektorientering. Det har støtte for å samle data i strukturer, men ingen språklig støtte for å knytte dette sammen med operasjoner. Det er mulig å bruke *static* på globale navn for å hindre linkeren å se navnet. Det er dog mulig å skrive objektorientert kode, selv om språket ikke er veldig til hjelp, og C er ikke altfor ille å skrive objektorientert kode i – man kan bruke typecasting og egenskapen om at strukturenes medlemmer kommer i samme rekkefølge, til å simulere arv,[2] samt at arrayer med funksjonspekere kan simulere polymorfi.

Det som egentlig er av interesse, er hva slags muligheter pythongrensesnittet gir. Alle Pythons objekter nås via strukturen *PyObject*, men hva er det? Et segment sakset fra *include/python/object.h*, gir svaret.

```
#define PyObject_HEAD                   \
        _PyObject_HEAD_EXTRA            \
        int ob_refcnt;                  \
        struct _typeobject *ob_type;

/* Nothing is actually declared to be a PyObject, but every pointer to
 * a Python object can be cast to a PyObject*.  This is inheritance built
 * by hand.  Similarly every pointer to a variable-size Python object can,
 * in addition, be cast to PyVarObject*.
 */
typedef struct _object {
        PyObject_HEAD
} PyObject;
```

(*_PyObject_HEAD_EXTRA* er vanligvis definert tom.) Dette er den samme muligheten for å simulere arv, som er beskrevet ovenfor. Python gir så brukeren flere funksjoner som tar en peker til *PyObjekt*, og gjør forskjellige ting med den. Grensesnittet søker i så måte å simulere objektorientering, men det er noen ulemper som kunne vært bedre i andre språk. En C++-programmerer vil gjerne tenke seg å skrive «peker->metode()» heller enn «funksjon(peker)» – å gjøre biblioteket mer samsvarende med språket, gjør bruken enklere og mer estetisk. Referansetelling er også noe en bruker ikke skulle trenge å bry seg om, men C mangler konstruktører og destruktører, så da blir det slik. Men ved å la grensesnittet bestå av C++-klasser i et hierarki, blir det enklere og penere, samt at man kan kapsle inn minnehåndteringen.[3]

[2] For eksempel Xlib*http://www.x.org/* gjør dette.
[3] Det finnes for eksempel søppelsamlere for C++ og C.

Unntakshåndtering (exception handling)

En språklig finesse både Python og C++ tilbyr, er unntakshåndtering. Når noe går galt, kan programmereren kaste et unntak, som er representert ved et objekt. I Python er dette objektet som regel en instans av en subklasse av *Exception*, og i C++ er det ofte en subklasse av *std::exception*, skjønt ingen av språkene krever overholdelse av dette, men Python kommer nok til å innføre regelen. Når et unntak er kastet, spinner programmet seg ut av funksjonene til det blir tatt i mot, eller til man kommer ut av hovedprogrammet, hvilket medfører at programmet avsluttes med en feilmelding til bruker.

Unntakshåndtering er en genial oppfinnelse, som tillater programmereren å skrive algoritmen uten å teste returverdier og globale objekter, som *errno*, for feil hver gang man kaller en funksjon, som er måten det gjøres på i C. C har også *setjmp()* og *longjmp()*, men disse er nærmest ubrukelige sammenlignet med unntak. Det fine med unntak, er at det ikke er nødvendig å gjøre normal programkjøring mindre effektivt, noe som i enkelte tilfeller kan få stor betydning. I C vil man i slike tilfeller ta i bruk *longjmp()* og *setjmp()*.

Pythons grensesnitt mot C definerer som regel at unntak oppstår ved retur av spesifikke verdier, som regel nullpekere. Dermed må man som regel sjekke returverdiene, og ved unntak kan man kalle andre funksjoner for å få informasjon om det kastede objektet, samt eventuelt ta det i mot eller returnere nullpekeren selv. Det ville gjøre veldig mye for samhandlingen mellom språkene om man kan takle unntak ved å bruke C++-ekvivalenten.

Standardbibliotek

«Python kommer med batteriene inkludert,» mens C++ også kommer med et standardbibliotek som rommer mye av det man trenger. Cs standardbibliotek er dog magrere, og språkkjernen kan heller ikke kalles fyldig; de fleste språk har for eksempel bedre strenghåndtering enn C har. C++ har også et standardbibliotek utenom hva C har. C++' standardbibliotek deles gjerne inn i følgende deler:

C Hele C-standardbiblioteket er inkludert i C++.

Iostream Dette biblioteket tilbyr et klassehierarki med *cin*, *cout*, *stringstream* og *fstream*. *Iostream*-biblioteket er bedre enn Cs IU-bibliotek ved at mulighetene for formattert IU er typesikre, samt at *iostream* lett lar seg utvide for brukerdefinerte klasser.

STL-container Denne delen inneholder klasser som inneholder mengder av data. Alle containerklassene er *template*-klasser, som har lignende grensesnitt. Som regel er klassene bygget for tilgang gjennom pekerliknende iteratorer, slik at man kan skrive generiske algoritmer som kan jobbe med STL-containere og også innebygde arrayer.

STL-algoritmer STL-algoritmene er, for det meste, en samling *template*-funksjoner. Disse funksjonene jobber i stor grad med iteratorer, og tilfører en god del operasjoner som brukerne trenger.

I tillegg har C++-programmerere tilgang til flere biblioteker, blant annet Boost (tilgjengelig fra *http://www.boost.org/*, som er av generell interesse, samt andre biblioteker.

Python har selvfølgelig også støtte for de fleste typer containere, og, som seg hør og bør i et språk med OO-støtte, er abstrakte datatyper (ADT) implementert som klasser. Python benytter seg i stor grad av iteratorer, som språket har en spesiell støtte for[4]. I en situasjon hvor en C++-programmerer skal jobbe sammen med en Python-programmerer, vil det for C++-programmereren føles naturlig å benytte seg av sine vante biblioteker og teknikker, samt at det for python programmereren vil føles bekvemt å skrive på den vanlige måten. Et ønske kunne da være å kunne bruke C++-containere i Python, samt at de oppfører seg som andre Python-containere.

1.2 Problem

Innledningen har allerede hintet om at C++ og Python er to ofte benyttede språk. I mange prosjekter er Python et godt språk å skrive i, og det store standardbiblioteket tilbyr det meste av hva man trenger. Dette kommer i kontrast til C++, hvor man gjerne må lete etter biblioteker man kan bruke. I tillegg har Python gjerne et høyere abstraksjonsnivå enn C++ over maskinvaren, og programmerere er gjerne mere produktive i Python enn i C++.

I [7] skriver Hans Petter Langtangen om hvorfor Python er et godt valg for *scientific programming*. For eksempel kan Python samarbeide med mange eksisterende verktøyer, og det er ofte mulig å bruke eksisterende, veltestet kode. I tillegg legger han vekt på Pythons brukbarhet for andre ting enn bare tallknusing, og spesielt grafiske brukergrensesnitt. Det er enkelt å teste ting i Pythons interaktive skall, og slike eksempler er ofte enkelt å klippe ut og vise andre.

Problemet med Python, er at tolkeren ikke utfører instruksjonene særlig effektivt. Sammenligner man, kommer man gjerne opp i faktorer over 100 (se eksemplet i figur 1.1). Dette er ikke et stort problem for store deler av de fleste prosjekter; i [6] presenterer Knuth Paretos prinsipp, som sier at 80 % av tiden tilbringes i 20 % av koden (også kalt 80–20-regelen).

Paretos prinsipp gjelder i stor grad store prosjekter, men også relativt små prosjekter drar nytte av det. Det er ofte slik at man får mest igjen for pengene ved først å skrive programmet, for deretter å finne kandidater for optimalisering ved å måle kjøretid.

Når det gjelder programmer skrevet i Python, kan det å optimalisere innebære å skrive deler i mere effektive språk. Det finnes også andre alternativer, for mange av bibliotekene som følger med Python er skrevet i C. Det å benytte innebygde funksjoner kan spare mye tid fremfor å skrive sine egne løkker.

Men ofte er alternativet å implementere deler i f.eks. C++. Det er stort sett greit, men det er gjerne knotete å følge grensesnittet mot Python. Å skrive sine egne C++-moduler fører ofte til krasj, gjerne på grunn av referansetelling og feil bruk av andre deler. Det er også tidkrevende å få det rett; dette er bortkastet bruk av programmererressurser.

Men hvis dette er bortkastet, så må det finnes bedre alternativer. Jeg vil her se på måter å få Python-programmer til å kjøre fortere. En metode for dette, kan være å kompilere Python-kode til C++. Av testen i figur 1.1, ser man at mye kan vinnes på nettopp dette.

[4] Jeg tenker i stor grad på det faktum at *for*-løkken bruker iteratorer.

Figur 1.1: Sammenligning av C++ og Python

```
def f(a): pass

def main():
    from time import clock

    t = clock()
    i = 0
    while i < 10000000:
        a = 5.0
        b = 3.7
        c = a*b*i
        f(c)
        i += 1

    t = clock() - t
    print t

main()
```

gir hos meg jevnt over en utskrift på om lag 17 sekunder. C++-«ekvivalenten»

```
#include <iostream>
#include <ctime>

void f(double);

int main()
{
    std::time_t t = std::clock();
    int i = 0;
    while (i < 10000000) {
        double a = 5.0;
        double b = 3.7;
        double c = a*b*i;
        f(c);
        ++i;
    }
    t = std::clock() - t;
    std::cout << double(t)/CLOCKS_PER_SEC << '\n';
}

void f(double d) {}
```

gir en utskrift på ca. $0,17$ sekunder. Eksemplet ble kompilert uten optimalisering.

2 Eksisterende systemer

Tanken om å optimalisere Python er ikke enestående, og andre har tenkt den. Dette synes i et stort antall verktøy for å hjelpe til med å skrive kjappere kode.

2.1 Sammenkobling av C++ og Python

Jeg skal riktignok se på muligheter for å kompilere Python til C++, men det er også interessant også å kjenne til løsninger som lar brukeren bruke C++-kode på en enklere måte enn hva Python selv tillater.

Det finnes et antall verktøy for å koble sammen C++- og Python-kode. De har forskjellige vinklinger, men felles er at de gjør koblingen enklere. Typisk vil disse verktøyene sørge for referansetelling, slik at man unngår minnelekkasjer og aksessering av frigitt minne. Verktøyene vil tilby konverteringer fra Pythons typesystem til C++', og vil dermed stort sett la brukeren bruke et enklere grensesnitt for kommunikasjon med Python-tolkeren.

2.1.1 SWIG

SWIG (*http://www.swig.org/*) er et program som, gitt grensesnittfiler, genererer en C-fil som bruker Pythons grensesnitt og kaller C- eller C++-kode. Grensesnittfilen vil ligne mest mulig på en header-fil, altså med C/C++-deklarasjoner[1] av de C/C++-funksjoner man trenger. Ut får man en fil som da oversetter Pythons argumenter til C-typer, og kaller de eksisterende funksjoner.

En typisk bruk av SWIG, innebærer å gi verktøyet en kopi av de allerede eksisterende headerfilene. Hver gang SWIG klager, må man så modifisere litt, inntil man får et brukbart resultat. Det er vel, ut fra dette, gitt at den enkleste måten å bruke verktøyet på, er å skrive programmet i Python, for så å benytte seg av rutiner skrevet i C.

Det er en grunn til at denne delen av dokumentet omhandler SWIG, og ikke wrappergeneratorer i alminnelighet. Grunnen til dette er at SWIG er ganske mye bedre enn de andre. Verktøyet kommer med en fullstendig parser for C og C++: det har støtte for ANSI C, og store deler av C++. På grunn av måten Python spesielt, og andre skriptespråk generelt, tilbyr C-grensesnitt, er det dog veldig vanskelig å ta i bruk alle C++' muligheter, ettersom dette ikke lar seg gjøre med C. Likevel takler SWIG vanskelige deler av C++ på en imponerende måte; det takler til og med templates, som iboende er vanskelige, ettersom de må instansieres av en kompilator, som må ha tilgang til kildekoden[2], for så

[1] Det er intet språk ved navn «C/C++». C og C++ er to distinkte språk, men SWIG kan jobbe med begge, og derfor skriver jeg C/C++.

[2] Man trodde kanskje en gang at så ikke var tilfelle, så C++ har et nøkkelord som skal kunne eksportere templates til andre kompileringsenheter (C++-sjargong for kildekodefil), men kun en kompilator

å kunne generere en binærfil. Det er dog en viktig begrensning: De aktuelle instansene må eksplisitt instansieres.

Ettersom SWIG støtter såpass mye, er det kanskje bedre å diskutere begrensningene.

C og C++ spesifiserer at arrayer, ved de fleste situasjoner, skal konverteres til pekere til arrayens første element. Spesielt viktig er at dette gjelder for argument- og returverdier. Dette vil ganske enkelt medføre at det er umulig å kjenne forskjell på arrayer og pekere til enkeltobjekter. Løsningen er å gjøre pekere om til typer Python ikke kan gjøre noe med. Det hadde kanskje vært veldig praktisk å muliggjøre at Python kan søke tilgang til enkeltelementer i medsendte C-arrayer, men isåfall må man skrive et passende grensesnitt selv.

SWIG takler på ett vis unntak. Dersom man bruker unntaksspesifikasjonslister i C++, vil SWIG automatisk generere en unntakssjekk, for så å kaste et tilsvarende unntak i Python. Det er dog veldig uvanlig å skrive unntaksspesifikasjonslister i C++, og det er flere gode grunner til dette. Å skrive unntaksspesifikasjoner er som regel ikke verdt bryet, men de kan fort være det når man skriver for SWIG. SWIG har også støtte for å spesifisere *try–catch*-blokker i funksjonswrapperne, så det er ikke en helt forferdelig oppgave.

Et potensielt problem med SWIG, er at det ikke tar hensyn til *namespace*. Det vil bli parset, men i Python vil alle symbolene kastes inn i roten av modulnavnerommet. Ettersom navnerom er C++' måte å implementere moduler på, følger det at man som regel lager en modul av ett navnerom, noe som i stor grad vil rettferdiggjøre dette valget.

Det er også mange andre småting man legger merke til. Småting som skyldes at C++ og Python er to forskjellige språk. SWIG tar en del veloverveide beslutninger som kanskje ikke ville vært god praksis ellers. Et eksempel: En funksjon returnerer en klasse, og ikke en peker. SWIG vil isåfall lage et nytt objekt ved hjelp av *new* (C++) eller *malloc()*, og returnere et objekt som holder pekeren. Er man ikke var på dette, vil dette fort føre til uheldige minnelekkasjer.

2.1.2 Boost.Python

Boost.Python (se *http://www.boost.org/*) er et C++-bibliotek som wrapper Pythons C-API med bruk av klasser og templates. Biblioteket utnytter dermed C++' egenskaper for å erstatte wrapperklasseskrivingen med funksjonskall.

SWIG er forskjellig fra Boost.Python i at SWIG genererer et grensesnitt med wrappere fra noe som ligner headerfiler, mens med Boost.Python er laget for å forenkle den manuelle skrivingen av grensesnitt.[3] Ut fra dette virker det ganske logisk at SWIG egner seg bedre til å generere Python-grensesnitt av eksisterende bibliteker, mens Boost.Python bør vurderes dersom man vil skrive nye programmer hvor språkene samhandler.

Biblioteket har flere kjerneområder. Den første og viktigste når man skriver kode, er at det har C++-wrappere for Python-objektene. Disse wrapperne er et stort klassehierarki, med *boost::python::object* som rot. Denne klassen har definert funksjoner og operatorer,

støtter dette, og den trenger fortsatt tilgang til kildekoden. I [9] og [10] diskuterer Sutter dette.

[3] Det finnes også et verktøy, «Pyste», som ytterligere kan forenkle wrappergenereringen; dette bruker dog omstendelige grensesnittfiler.

slik at det er trivelig å jobbe med den i C++. For eksempel er en Python-funksjon et objekt, og den kan eksekveres ved å kalle *object::operator()()*.

Som sagt, må man selv definere grensesnittene. Dette gjøres ved å bruke konstruksjonen *def("pynavn", funksjon)* for å definere funksjonen *funksjon()* under navnet *pynavn()* i Python. Templates tar seg av argument- og returtyper. Det finnes også versjoner av *def()* som tar tre og fire argumenter. De to siste er da henholdsvis argumentnavnene, slik at keyword-argumentene virker i Python, og doc-teksten.

Klasser defineres ved å bruke klassen *class_*. Denne har *def()* som medlemsfunksjon, og denne vil definere medlemsfunksjoner på samme måte som det tilsvarende ikke-medlemmet.

Alt i alt: Med Boost.Python ser grensesnittet pent ut.

Dessverre er aldri livet en dans på roser. Boost.Python er som alltid begrenset av Cs muligheter, og ved å se på dokumentasjonen, obsen[4] og todo-listen, kan man se at en del mangler. Jeg har dog ikke fått testet biblioteket skikkelig ennå, så jeg kan ikke komme med en subjektiv vurdering her.

2.2 Optimalisering av Python-kode

Det er også gjort flere forsøk på å få Python til å eksekvere kjappere. Jeg vil her gå inn på noen av dem.

2.2.1 Psyco

Psyco er en JIT-kompilator for Python. Den kan lastes ned fra *http://psyco.sourceforge.net/* med full kildekode. Psyco virker ved at den oversetter dynamisk Python-bytecode til assemblerkode. Potensielt oversetter den en funksjon med flere versjoner – eller spesialiseringer – og velger under kjøring en effektiv versjon. Denne oversettelsen foregår under kjøring, og den kan lage nye spesialiseringer når den ser kode.

Denne oversetteren kommer virkelig til sin rett når koden er spesielt algoritmisk. Kanskje en løkkestruktur og bearbeiding av enkle typer, og man kan i beste fall vente opp til hundre ganger mere effektiv kode. For mindre ideale forhold vil man fortsatt ha dobbelt så rask kode som Python kan utføre.

I bruk er Psyco veldig enkel. For mindre script kan man bare legge inn linjene

```
try:
    import psyco
    psyco.full()
except:
    pass
```

De to linjene inne i *try*-blokken er alt som egentlig trengs. Hvis Psyco ikke er installert, vil *try*-konstruksjonen sørge for at programmet fortsetter som normalt (uten optimalisering). Ut over dette er det ikke nødvendig å forandre programmet for å benytte Psyco.

[4] obs=FAQ

Psyco har sine ulemper også. For det første genereres assemblerkode. Assemblerkode for 32-bits Intel $x86$-maskiner. For lesere som ikke har vært borti assemblerspråk, kan jeg fortelle at hver prosessortype har sitt eget. Assemblerkode for 32-bits $x86$ kan oversettes til maskinkode for denne prosessortypen, og den eneste måten å kjøre koden med en annen maskin, er å bruke en emulator. Nå er ikke $x86$ en uvanlig prossesortype. De fleste PC-er benytter denne prosessoren (eller en annen prosessor som benytter samme maskinkode), og til og med nyere Apple-maskiner benytter denne prosessoren. Men man kan spesifikt ikke kjøre på Sparc-prosessorer, som Sun-maskiner kommer utstyrt med. Et annet problem med assemblerkode, er at den må utføre funksjonskall manuelt. Dvs. at koden må implementere hvordan argumenter sendes til funksjonen, hvordan returverdien mottas og alt annet på samme måte som C-kompilatoren gjør. Stort sett virker dette på samme måte med forskjellige C-kompilatorer på samme maskin, men ikke alltid. Av Psycos nettsider, kan man se at Psyco ikke fungerer på Mac OS X uten å endre noen detaljer omkring funksjonskall.

En annen ulempe er at Psyco bruker *mye* minne. Manualen foreslår at man for annet enn de mindre prosjektene ikke benytter psyco. full (), som optimaliserer *alle* funksjonene i programmet. Psyco har flere funksjoner for utvelgelse av funksjoner for optimalisering. Den enkleste er nok psyco. profile (), som bruker profileringsteknikker for å avgjøre hva som bør optimaliseres.

Likevel har Psyco mye for seg. Ved å legge til linjene ovenfor i Python-programmet i figur 1.1, ble utskriften jevnt over 0.38, altså mye nærmere C++-versjonen enn Python-versjonen, og om lag 2,2 % av Python uten Psyco.

2.2.2 Pyrex

Pyrex er et språk som minner om Python, men det har utvidelser som typedeklarering. Språket er designet for å skrive extension modules, dvs. kompilerte moduler.

Den offisielle Pyrex-kompilatoren oversetter til C-kode, som kan kompileres til en extension module. Utvidelsene i Pyrex går derfor ut på å guide Pyrex' oversettelse til C-kode. Utvidelsene er slikt som: Deklarering av variable med C-typer, deklarering av C-funksjoner, definering av C-strukturer, enums, unioner.

Pyrex vet en del om konvertering av datatyper fra Python til C og tilbake igjen. I tillegg tilbys typecasting mellom ulike C-typer. Dette gjør at Pyrex vet en del om datatypene i funksjonene som kompileres, og på bakgrunn av dette, kan Pyrex generere effektiv C-kode. Hvis man ikke deklarerer variable, vil Pyrex anta at de har typen object, noe som innebærer at de kan brukes, men med en svakere kjøretid.

Ulempen med Pyrex, er at det er et eget språk. I veldig mange tilfeller kan man kompilere ren Python-kode, men dette vil da ikke dra full nytte av Pyrex' egenskaper. Med Pyrex, kjører eksemplet i 1.1 på 14 sekunder. Ved å sette inn deklarasjoner av alle typer, går kjøretiden ned i 0.19 sekunder.

2.2.3 Shed Skin

Shed Skin er utviklet som et prosjekt for å finne ut om man kan lage en Python-kompilator som foretar typededuksjon. Hvordan den virker forklares relativt grundig i [2]. Kort fortalt benytter den seg av Ole Agesens *Cartesian Product Algorithm*[1].

Shed Skin er en kompilator. Den lager C++-programmer av statisk typede Python-programmer. Shed Skin kan lastes ned fra *http://sourceforge.net/projects/shedskin/*.

Siden første utgave har nå Shed Skin fått mulighet for å generere extension modules. Denne støtten kom først med versjon 0.0.22, som ble sluppet i juni 2007.

Kompilatoren gjør en stødig jobb med koden, og resultatet er stort sett lesbart, noe som er praktisk når man vil finne feil eller hvis man vil se over resultatet.

I forhold til Pyrex-brukere, kan Shed Skin-brukere glede seg over at kompilatoren kompilerer ren Python-kode til C++. Det er riktignok en del begrensninger, men man trenger ikke lære og bruke et helt nytt språk.

Ulempen med Shed Skin, er det begrensede utvalget biblioteker. Shed Skin tar sikte på å skille seg ad fra tolkeren, og da finnes det ingen annen måte å støtte bibliotekene enn å implementere dem i C++. Alternativet ville være å benytte Python-bibliotekene og Pythons typer til intern representasjon. I tillegg til det noe begrensede antall biblioteker, kommer noen restriksjoner på koden. Den viktigste er at alle variable må være statisk typet, det vil si at hver variabel bare kan holde en type.

```
def foo():
    a = 0  # a is an int
    for i in range(100): # i is an int
        a = 3.0*i + 5.0*a  # 3.0*i + 5.0*a is a float
    return a # int or float?
```

Dette vises av eksemplet. Her er *a* først et heltall fordi høyresiden i første linje er 0, et heltall. I løkken er høyresiden et flyttall. Et menneske ville slått fast at *a* bør være et flyttall. Men dette er fordi et menneske kan se intensjonen: Ettersom *a* evalueres i løkken, må den ha en initiell verdi, og 0 og 0.0 er begge null. I tillegg fungerer dette i Python fordi *a* simpelthen blir erstattet i alle tilordningene. Dessverre krever dette intelligens, noe et datamaskinprogram ikke har. Skjønt det går an å informere kompilatoren om enkelttilfeller som dette, finnes det ingen generell måte å ordne problemet.

Eksemplet i figur 1.1 ble kompilert og kjørt med Shed Skin. Shed Skin virker slik at man kjører programmet på hovedfilen. Ut fra dette lager Shed Skin C++-filer, samt en makefile, som gjør at man enkelt kan bruke programmet make for å kompilere og lenke filene. Vanligvis er det en enkel greie, men for akkurat dette eksemplet er det vitalt ikke å bruke optimalisering i C++-kompilatoren (GNU-kompilatoren optimaliserer bort hele løkken, som den vet ikke gjør noe som helst.) Like interessant som kjøretiden er kanskje kompileringstiden. Shed Skin brukte 8,6 s[5] og g++ brukte 15,6 s på å kompilere dette lille programmet. Programmet i seg selv brukte 0,19 s, noe som for alle praktiske formål er like godt som C++-versjonen.

[5] Tid tatt med time(1), og avlest *user*

Figur 2.1: Figur 1.1 for Shed Skin og Pyrex

For å kompilere med Shed Skin, måtte jeg flytte *import*-setningen til toppen:

```
from time import clock

def f(a): pass

def main():
    t = clock()
    i = 0
    while i < 10000000:
        a = 5.0
        b = 3.7
        c = a*b*i
        f(c)
        i += 1

    t = clock() - t
    print t

main()
```

Her er alt deklarert for Pyrex:

```
cdef void f(double a):
    pass

def main():
    from time import clock

    cdef int i
    cdef double a, b, c
    cdef double t
    t = clock()
    i = 0
    while i < 10000000:
        a = 5.0
        b = 3.7
        c = a*b*i
        f(c)
        i = i + 1

    t = clock() - t
    print t
```

3 Grunnlag for design

Hensikten bak prosjektet er å kunne kjøre rutiner skrevet i Python raskere enn idag. Dette gjelder spesielt numeriske rutiner, såkalt tallknusing. For å oppnå dette, er det ønskelig å bruke et kompilert språk. Spesielt kommer dette kravet fra

Ettersom Python har et C-grensesnitt, som både C og C++-kompilatorer kan benytte seg av, ville det være en fordel om Python-rutinene var skrevet i C eller C++. I tillegg er det interessant å kunne bruke Python-moduler, som for eksempel NumPy og dens ndarray-klasse. For dette prosjektet kan NumPy-støtte karakteriseres som essensielt, fordi dette vil gjøre det mulig å bruke NumPy-arrayer effektivt også der koden ikke kan benytte de optimaliserte rutinene i NumPy-pakken.

På bakgrunn av dette, er prosjektet å implementere en Python-kompilator. Kompilatoren skal oversette Python-kode til C++. Det er også et mål i seg selv at kildekoden skal være lesbar for mennesker, og dette poenget er viktigere enn å implementere alle sider av Python. Derfor har jeg kuttet ut biter som ikke enkelt kan uttrykkes i C++. Eksempler på dette er ting som *yield*.

I dette kapitlet vil jeg forklare prinsippene bak kompilatorens design. En del valg, som avveier troskap mot Python-spesifikasjonen mot andre mål, må åpenbart gjøres, og dette kapitlet vil argumentere for valgene som er gjort.

3.1 Hva må støttes

La oss ta noen eksempler. For alle eksemplene antar jeg at disse to linjene eksisterer:

```
import numpy
import math
```

Eller med andre ord: Modulene *numpy* og *math* kan brukes.

```
def foo(a, b, c):
    for i in range(10 - 5):
        print i
    print a + b + c
    return a
```

Kompilering av denne til C++ medfører en rekke spørsmål. For det første hva er symbolene *a*, *b* og *c*? Det finnes to muligheter: a) instanser av en generell *object*-klasse, b) brukeren må deklarere dem, eller c) kompilatoren kan gjette basert på hvordan objektene brukes.

Python har valgt *a*. Alle objekter har typen *object*, eller for C-implementasjonens del: alt er pekere til strukturen *PyObject*. Alle operasjoner på objektene gjøres ved å finne

rett funksjon i en virtuell tabell. For de innebygde operatorer på innebygde typer, er dette snakk om noen oppslag i C-arrayer, mens det for vanlige funksjoner skrevet i C innebærer å slå opp funksjonen i en hash-map. Når så operatorens funksjon er kalt, må den så finne ut av hvilken type den andre operanden har. Ved å gjøre om operasjonene til kall på virtuelle medlemsfunksjoner vil man få C++-kompilatoren til å skrive nøyaktig det C-implementasjonen allerede gjør: *PyObject* inneholder en peker til en typebeskrivelsestruktur som holder arrayer til funksjoner til operatorene. I C++ er dette implementert ved at kompilatoren setter inn en ekstra peker i klassene, og denne pekeren peker til en kompilatorgenerert typeinformasjonsstruktur, som, i tillegg til virtuelle kall, brukes av *dynamic_cast* og *typeid*.

Å velge *a* vil gjøre det mulig å implementere hele python på en relativt mye enklere måte enn noe annet. I tillegg vil koden enkelt kunne samhandle med Python: Alt som trengs er en map fra C++-kompilatorens typeinformasjonsstruktur til Pythons typeinformasjonsstruktur og tilbake og funksjoner som kan konvertere. Dette er f.eks. en viktig del av hvordan programmet biblioteket *Boost.Python* virker.

Men C++ har et kraftig typesystem, og bruk av det vil gjøre koden raskere. Istedenfor den lange sekvensen ovenfor, kan operasjoner som heltallsaddisjon erstattes med nettopp dette. Vet kompilatoren på forhånd at typene er *int*, betyr dette mye kjappere kode. Til gjengjeld ofrer man noe: Det er nødvendig å avgjøre hvilke typer objektene har på forhånd. Dette må gjøres før kodegenerering.

La oss så se på de andre alternativene. Alternativ *b* er enkel: Før *foo* kan konverteres til C++, må brukeren av kompilatoren fortelle Python hvilke typer som forventes. Man må kunne anta at brukeren hadde noe spesielt i tankene da han skrev funksjonen, og ved å deklarere typene selv, vil han kunne oppdage feil som ellers ikke hadde blitt oppdaget med alternativ *a*. Ulempen med dette er at det ofte kan være mulig å gjette etter bruk, og hvis kompilatoren kan gjette, er det ikke fornuftig at programmereren bruker tid på det.

Alternativ *c* høres derfor fornuftig ut. Spørsmålet er om det er mulig. Tar man eksemplet ovenfor, kreves det av argumentene at de støtter operatoren + og konvertering til streng. Det er få typer i Python som ikke støtter dette. Alternativet kan likevel bli enklere hvis man definerer en dragning mot bestemte typer. Jeg har her valgt å benytte typene *float* (tilsvarer *double* i C++) og NumPys *ndarray* av *float*. Kompilatoren søker å benytte *float*, men *ndarray* benyttes hvis variabelen er indeksert.

Hvis vi nå søker disse typene, hva vil *foo* se ut som i C++? For å demonstrere dette, så har jeg oversatt koden til C++ for hånd.

```
double foo(double a, double b, double c)
{
    for (int i = 0; i < 10-5; ++i)
        std::cout << i << '\n';
    std::cout << a + b + c << '\n';
    return a;
}
```

Her ser man flere interessant ting: *For*-løkken i Python er blitt en *for*-løkke i C++. Dette krever informasjon om funksjonen *range*. I Python returnerer *range* en liste med heltallsverdier, og *for*-løkken setter *i* til ett av elementene for hver iterasjon. I det hele tatt er Pythons *for*-løkke noe helt annet enn hva C++'. Jeg har valgt å oversette *range* spesielt fordi den er mye brukt, og fordi da C++ har et ekvivalent idiom, som også vil sørge for raskere kode. At *print*-setningene er oversatt til bruk av *cout* er naturlig. Kompilatoren har også gjettet at *foo*s resultattype er *double*, ettersom den eneste *return*-setningen returnerer *a*, som er antatt Python-*float*.

Dette eksemplet bryter også med Pythons regler:

```
a = 3.14
b = foo(a, a, a)
print a is b
```

gir følgende utskrift:

```
0
1
2
3
4
9.42
True
```

Funksjonen som kalles er i dette eksemplet den opprinnelige Python-implementasjonen. Den interessante utskriften er den nederste linjen. Testen a **is** b, som sammenligner verdiene til de underliggende *PyObject*-pekerne, evalueres sann, og en korrekt oversettelse til C++ hadde måttet ta hensyn til det. I det generelle tilfellet måtte man ha sendt argumentene som pekere[1] og returnert resultatet som en peker. I tillegg hadde det vært nødvendig å sjekke resultatets adresse mot alle inn-adresser, for så å finne en sammenheng. Dette vil ikke bare medføre ekstraarbeide i forbindelse med funksjonskall, men også medføre at funksjonen blir mindre brukbar i andre sammenhenger: Slik den oversatte funksjonen ser ut, kan den med letthet settes inn i et C++-program.

I akkurat dette tilfellet hadde en analyse kunnet avsløre at returvariabelen alltid er argumentet «a», men å gjøre slike analyser er litt for mye å kreve av en masteroppgave.

Her er nok et eksempel:

```
def bar(a, b, c):
    sum = 0.0
    for i in range(0, len(a)):
        sum += a[i] + b[i] + c[i]
    return sum
```

I dette eksemplet er alle argumentene indeksert. Følgelig skal typen deres være NumPy-array av *float* og dimensjon én. I tillegg er det her definert en lokal variabel, *sum*. Det er ganske uproblematisk at den har type *float*. La meg oversette til C++:

[1] Referanser er i denne sammenheng også pekere

```
double bar(ndarray<double, 1> a, ndarray<double, 1> b, ndarray<double, 1> c)
{
    double sum = 0.0;
    for (int i = 0; i < len(a); ++i)
        sum += a(i) + b(i) + c(i);
    return sum;
}
```

Det interessante her er navnene: *ndarray* og *len*. Hva gjør de, og hvordan fungerer de? Det var tidlig en antagelse at å skrive en *ndarray*-klasse ville spare arbeid. Alternativet ville være å sende med pekere:

```
double bar(double a[], std::size_t len_a,
           double b[], std::size_t len_b,
           double c[], std::size_t len_c)
{
    double sum = 0.0;
    for (int i = 0; i < len_a; ++i)
        sum += a[i] + b[i] + c[i];
    return sum;
}
```

Fordelen med å skrive en array-klasse, er at den kan innkapsle andre operasjoner man gjerne måtte ønske. For eksempel takler klassen array-operasjoner som snarveier. Arrayen kan også sjekke indekseringer, noe som kan slås av med ulike opsjoner til C++-kompilatoren.

Selvfølgelig er det ønskelig at andre funksjoner enn *len* og *range* skal fungere:

```
def baz(a, b, c):
        sum = 0.0
        for i in range(0, len(a)):
            sum += math.cos(a[i]) + math.pow(b[i],3.7) + c[i]
        return sum
```

bør oversettes i retning av:

```
double baz(ndarray<double,1> a, ndarray<double,1> b, ndarray<double,1> c)
{
    double sum = 0.0;
    for (int i = 0; i < len(a); ++i)
        sum += std::cos(a(i)) + std::pow(b(i), 3.7) + c(i);
    return sum;
}
```

Ser man på det semantiske, er det også problemer med denne løsningen. I Python er modulene i virkeligheten objekter. En teknisk korrekt løsning burde derfor vært mere i retning av

```
object *math = new module("math");
```

```
double baz(ndarray<double,1> a, ndarray<double,1> b, ndarray<double,1> c)
{
    double sum = 0.0;
    for (int i = 0; i < len(a); ++i)
        sum += math["cos"](a(i)) + math["pow"](b(i), 3.7) + c(i);
    return sum;
}
```

Dette er nettopp de dynamiske strukturene jeg vil unngå. I tillegg må returverdiene fra *object::operator[]* kunne konverteres til noe som kan kalles med forskjellige antall argumenter. Når dette først er gjort, er det heller ingen generell måte å informere kompilatoren om at disse kallene returnerer en noe som kan legges sammen med en *double* og gi et resultat som kan konverteres til *double*. Likevel er modulene så integrert i Python at det ville være unaturlig å forby dem. Men støtte for moduler må åpenbart begrenses i forhold til hva Python tilbyr. Jeg vil behandle dette grundigere i seksjon 4.6.

Skjønt strenger er innebygget i Python, er det ikke essensielt at de skal fungere på nøyaktig samme måte som i Python. Det viktigste med strenger, er å kunne skrive dem ut.

```
def moo(a, b, c):
    sum = 0.0
    for i in range(0, len(a)):
        print "a[%d]=%e" % (i, a[i])
        sum += a[i] + b[i] + c[i]
    return sum
```

Her er et eksempel på kode: Ikke bare skal strengen skrives ut, men den skal også inngå i et uttrykk som returnerer en annen streng. Det er flere måter å takle dette på. For eksempel kan man gjenkjenne **print** "format"\% argument. Det vil si: når formateringsuttrykket står i klartekst og skal skrives ut, kan man gå gjennom det helt statisk:

```
double moo(ndarray<double,1> a, ndarray<double,1> b, ndarray<double,1> c)
{
    double sum = 0.0;
    for (int i = 0; i < len(a); ++i) {
        std::cout << "a[" << i << "]=" << std::scientific << a(i);
        sum += a(i) + b(i) + c(i);
    }
    return sum;
}
```

Som man ser: strengen er kuttet opp, og argumentene er skrevet ut der prosentformateringene stod. Kompilatoren kjenner typen til alle argumentene i tuppelen, og hvordan de forskjellige formatflaggene virker, kan effektivt behandles av kompilatoren.

Løsningen over er god, men det føles ikke riktig ikke å kunne støtte annet enn basale formateringer i *print*-setninger. Det bør ikke være noe stort problem å kunne støtte

forateringer i andre sammenhenger. Et nytt forsøk benytter seg av *sprintf*/*snprintf*[2]:

```
double moo(ndarray<double,1> a, ndarray<double,1> b, ndarray<double,1> c)
{
    double sum = 0.0;
    char tmp[N];
    for (int i = 0; i < len(a); ++i) {
        snprintf(tmp, N, "a[%d]=%e\n", i, a(i));
        std::cout << tmp;
        sum += a(i) + b(i) + c(i);
    }
    return sum;
}
```

Alle som har skrevet kode i C, vil kjenne igjen Pythons formateringssyntaks: Den er med noen små endringer lånt fra *printf*-familien i Cs standardbibliotek. I eksemplet ovenfor lagres resultate til en midlertidig variabel, *tmp*. Denne er en vanlig array. Størrelsen N kan enten være «stor nok», eller så kan kompilatoren prøve å beregne plassen. I dette tilfellet, kan man også bruke *printf* direkte:

```
double moo(ndarray<double,1> a, ndarray<double,1> b, ndarray<double,1> c)
{
    double sum = 0.0;
    for (int i = 0; i < len(a); ++i) {
        std::printf("a[%d]=%e\n", i, a(i));
        sum += a(i) + b(i) + c(i);
    }
    return sum;
}
```

Denne har igjen samme problem som det første forsøket: Den gjelder bare utskrift. Den deler også et problem med *snprintf*-varianten: Den er ikke typesikker. Dette betyr at kompilatoren *må* validere formatstrengen mot argumenttypene.

Et annet alternativ er å skrive formateringsrutinen i C++, og la kompilatoren generere kall til denne. I fremtiden vil det også bli mulig å skrive *printf*-lignende saker typesikkert i C++[3].[3] Idag kan man gjøre som Python: la operanden være en *tuple*, men jeg vil vente litt med å eksemplifisere det.

```
double moo(ndarray<double,1> a, ndarray<double,1> b, ndarray<double,1> c)
{
    double sum = 0.0;
    for (int i = 0; i < len(a); ++i) {
        std::cout << format("a[%d]=%e\n", i, a(i));
        sum += a(i) + b(i) + c(i);
```

[2] *snprintf* er ny i 1999-versjonen av C. Av den grunn er den ikke nevnt i C++-standarden, men mange implementasjoner vil kunne tilby den.

[3] Med GNUs C++0x-kompilator kan det også gjøres idag.

```
    }
    return sum;
}
```

Hva er isåfall returverdien fra *format*? Det enkleste er ofte det beste, og *std::string* er ikke bare enkelt, men den vil også finnes på alle C++-kompilatorer. Ettersom utskrift likevel er relativt tregt, har jeg kommet til at det er like greit om ikke alle strenger oversettes med *std::string* (eller *std::wstring* for *unicode*-objekter).

Det er riktignok et problem med Unicode[11]: Det finnes ingen garanti for at Unicode-kodinger kan brukes. *wchar_t* er i C++-standarden en implementasjonsdefinert størrelse, og hvilket tegnsett den støtter, finnes ingen garanti for. Men med tanke på hvilke plattformer Python kjører på, tar jeg heller chansen enn å finne et passende bibliotek.

```
def popo():
    b, c = (2, 5)
    a = 5
    return a
```

3.2 Begrensninger

Python er et veldig dynamisk språk i flere aspekter enn typer. Hvis man skal få Python til å generere *effektiv* kode, er det naturlig at noe utelates. Ikke bare fordi det er ineffektivt i seg selv, men også fordi det isåfall går på bekostning av de ønsker som allerede er skissert.

I C++ kan man ikke slette symboler etter ønske. Når navnet går ut av skop, slettes objektet det forbindes med. I Python er dette annerledes: *Del* kan slette symboler fra symboltabellen. Hvis man hadde gått med på at alle symboler er pekere til dynamiske objekter, hadde dette tilsvart å sette selve pekerverdien til *null* i C++. Ettersom et av designønskene er at symboler ikke er pekere til dynamiske objekter, kan ikke *del* brukes på samme måte i C++. I Python kan også *del* brukes til å slette attributter og elementer fra objekter. Dette kan implementeres hvis attributtene og elementene er kjøretidsavhengige ressurser hos objektene. For eksempel ville sletting av et element i en Boost-tuppel medført endring av typen, og det er ikke mulig. Sletting av en medlemsfunksjon hos en innebygget type ville heller ikke mulig, men på dette området gir Python en kjøretidsfeil. Dette vil da endres til en kompileringsfeil.

Bruk av *exec* og også andre dynamiske funksjoner, som *eval*, fordrer kall til kompilatoren. Dette er ofte gjort sammen med *vars*, *globals* og *locals* som genererer mappinger tilsvarende symboltabellen. Dette er en av de typiske funksjonene som skiller kompilerte og tolkede språk[4], og det vil være en stor jobb å støtte bare deler av dette av hva *exec* tilbyr.

Jeg har allerede nevnt *yield*. Dette nøkkelordet støtter noe som ligner hva Knuth kaller *coroutines*. I [5, Seksjon 1.4.2] skriver Knuth:

[4] Skjønt flere tolkede språk, deriblant Python, kompilerer koden før den eksekveres.

> Subroutines are more general program components, called *coroutines*. In contrast to the unsymmetric relationship between a main routine and a subroutine, there is a complete symmetry between coroutines, which *call on each other*.

Poenget er at subrutiner kaller tilbake til den kallende rutinen når den er ferdig. Både Python og C++ har støtte for dette: Et funksjonskall lagrer returadressen før funksjonen kalles, og *return* sørger for å hoppe tilbake til den lagrede adresse. I Python vil *yield* også sørge for å lagre adressen, slik at rutinen kan kalles tilbake der den stoppet. I C++ finnes det ingen generell metode for å hoppe til et spesielt sted inne i en funksjon.

Det finnes dog en åpenbar algoritme for oversetting av *yield*-funksjoner:

```
def gen(end):
    for i in range(end):
        yield i
```

Kan oversettes med

```
class gen {
    struct gendata { //local variables in gen()
        std::vector<int> tmp1;
        std::vector<int>::iterator tmp2;
        int i;
        int end;
    };
    //states: one for every yield plus a start and end state
    enum states { s0, s1, send };

    gendata *p; //p == NULL iff state == send;
    states state;

    //end constructor
    gen() : p(NULL), state(send) {}

    public:

    //make sure *g++ is an int for g: gen
    class Int {
        int i;
        public:
        Int(int i) : i(i) {}
        int operator*() {return i;}
    };

    //exhausted iterator
    static gen end() { return gen(); }
```

```
    //call to gen() as in Python.
    explicit gen(int end) : p(new gendata), state(s0)
    {
        p->end = end;
        ++*this;
    }

    gen(const gen& o) : p(o->p?new gendata(*o.p):NULL), state(o.state) {}
    gen& operator=(const gen& o)
    { if (p) delete p; p = new gendata(*o.p); state = o.state; return *this; }

    int operator*() { return p->i; }

    gen& operator++()
    {
        if (state == s1)
            goto state1;
        if (state == send)
            return *this;
        p->tmp1 = range(p->end);
        for (p->tmp2 = p->tmp1.begin(); p->tmp2 != p->tmp1.end(); ++p->tmp2) {
            p->i = *p->tmp2;
            p->state = s1;
            return *this;
    state1:     ;
        }
        delete p;
        p = NULL;
        state = send;
        return *this;
    }

    Int operator++(int) { Int i(p->i); ++*this; return i; }
    friend bool operator!=(const gen&, const gen&);
};

bool operator!=(const gen& l, const gen& r)
{
    if (l.state == gen::send && r.state == gen::send)
        return false;
    else if (l.state != r.state)
        return true;
    else
```

```
        return l.p != r.p;
}
bool operator==(const gen& l, const gen& r)
{ return !(l != r); }
```

Som man ser blir dette relativt komplisert. Strukturen gendata inneholder alle lokale variable fra den opprinnelige funksjonen. Variablene *tmp1* og *tmp2* er i Python skjult, men også Python vil bruke midlertidige variable ved utførelse av *for*-løkker. Innmaten i den opprinnelige funksjonen er stappet inn i *operator++*, mens *operator** returnerer selve verdien. For å hoppe til rett plass i funksjonen, har jeg en spesiell tilstandsvariabel. Alle *yield* setninger oversettes med oppdatering av denne etterfulgt av en return. I tillegg finnes spesielle tilstander for begynnelse og slutt. Dette for å oversette Pythons idiomatiske iteratorer til C++-ekvivalenten, som er beskrevet som *output iterator* i [4][5]. Det ville ikke blitt veldig mye enklere hvis man skulle laget et iteratorobjekt som minner mer om Pythons (*next()*-medlem som slenger en *StopIteration* ved slutt. Ettersom jeg regner det å skrive en generator for såpass sjeldent, og fordi det er såpass mye arbeid, utelater jeg det.

Andre spørsmål er hvordan man skal støtte innebygde Python-typer. Med innebygde Python-typer, mener jeg de som ikke har direkte støtte i en motsvarende C++-type. Spesielt tenker jeg da på *dict*, *list* og *tuple*.

Ettersom alternativ a er forkastet, er det ikke trivielt å sette inn forskjellige typer samtidig som man skal ivareta typenes sekvenskrav:

```
def foo():
    for i in (5, 3.14, 'foobar', None):
        print i
```

Hvis i her skal ha en type, hva skulle det være? Det er flere mulige svar. For eksempel kan man rulle ut løkken:

```
void foo()
{
    {
        int i = 5;
        std::cout << i << '\n';
    }
    {
        double i = 3.14;
        std::cout << i << '\n';
    }
    {
        std::string i = "foobar";
        std::cout << i << '\n';
    }
    NoneType i = None;
```

[5] Merk at *output iterator* krever at *operator->* finnes hvis det er meningsfullt

```
    std::cout << i << '\n';
}
```

Dessverre er dette komplisert. I Python overlever løkkevariabelen løkken, slik at den kan benyttes etter løkken. Dersom løkken benytter seg av *break* eller *goto*, betyr det masse trøbbel, for i C++ kan man ikke hoppe over en initialisering. Det kompliserer også kompilatoren å legge inn støtte for at i kan ha forskjellige typer.

Jeg synes det er en helt grei begrensning at lister og tupler må holde seg til én type. Noen ganger kan det likevel være greit å ha tupler med forskjellige typer. I C++ kan man meget greit bruke Boosts Tuple-bibliotek for dette.

Den samme begrensningen som for lister må også gjelde *dict*-objekter: C++-kompilatoren må kjenne typen til d[ind].

En annen problematisk ting er Pythons støtte for variabelt antall argumenter og nøkkelordargumenter, det vil si støtte for konstruksjoner som ∗args og ∗∗kw. Grunnen er ikke først og fremst at dette er vanskelig å få til i C++, men heller det at funksjonen som mottar argumentene får disse pakket inn som henholdsvis *tuple* og *dict*. Dette blir vanskelig i C++.

3.3 Eksterne C- og C++-biblioteker

Ettersom oversetteren oversetter fra Python til C++, er det ønskelig å ta ibruk eksisterende C++-biblioteker. For eksempel kan man da kompilere kode som bruker biblioteker skrevet i C/C++, og hvis grensesnitt er gjort tilgjengelig for Python, for eksempel vha SWIG.

4 Bruk av systemet

Dette kapitlet vil introdusere kompilatorens brukergrensesnitt. Dette inkluderer hvordan man kontrollerer selve kompilatoren, samt hvordan man kan legge til funksjoner til kompilatorens bibliotek.

Gitt et program kompilatoren kan takle, vil kompilatoren generere to filer. Den ene filen er skrevet i ren C++, og inneholder definisjonen av de kompilerte funksjoner. Den andre filen inneholder wrapperfunksjoner som kan kompileres til en biblioteksfil som kan brukes av Pythons *import*. På denne måten er det enkelt å bruke funksjonene med og uten Python.

4.1 Installasjon

Systemet kommer i en pakke, og den standardiserte *setup.py* finnes i rotmappen. Dette skriptet benytter seg av *distutils*, som følger med Pythons standardbibliotek, og jeg vil ikke gå nærmere inn på hvordan *distutils* fungerer.

Kompilatoren distribueres i flere deler. Katalogen *pyemc* inneholder pakken ved samme navn, og denne pakken inneholder koden som kompilerer Python til C++. For å kompilere den resulterende C++-koden, trengs C++-kildekode som ligger i katalogen *include*. Disse filene inneholder C++-implementasjoner av diverse Python-funksjoner. I tillegg følger skriptet *pycompiler.py* med.

I pakken *pyemc* ligger modulen *config*. Denne er implementert i Python, og inneholder diverse standardvalg kompilatoren trenger. Filen er i seg selv dokumentert, men en installasjon vil trenge å oppdatere denne til lokale forhold. Kompilatoren kan i seg selv brukes uten annen installasjon enn å pakke ut filene og redigere *pyemc/config.py*, så lenge Python kan finne pakken *pyemc*, noe som kan gjøres ved å legge til rotmappen i *PYTHONPATH*.

4.2 Fra kommandolinjen

For enkel bruk av systemet, kan man bruke skriptet *pycompiler.py*. Dette skriptet tar en kildekodefil og et modulnavn som argument. Hvis alt går i orden genereres modulen hvis navn er gitt ved argumentet.

4.3 Som modul

Kompilatoren kommer i pakken *pyemc*[1]. Modulen *pyemc.pyc* inneholder klassen *Pyc*, som inneholder metoder for å gå gjennom kompileringen. Ved bruk av denne klassen kan man gå inn og definere parametertyper og returtyper. Ja, til og med typene til variable i funksjonen kan defineres. Faktisk bruker også kommandolinjescriptet denne klassen selv. Klassen *Pyc* har en *__init__*-metode som tar to strenger som parametre: Den første er kildekoden som skal kompileres. Den annen er et modulnavn. Dette navnet må være et gyldig navn i C++ og Python. Kompilatoren bruker dette navnet for å lage et navnerom[2] som inneholder C++-funksjonene, samt at det brukes for å generere en modul ved samme navn: Funksjonen *init<navn>* defineres for å initiere modulen, og den kompilerte filen kalles opp etter modulnavnet.

4.4 Kompileringsprosessen

Kompilatoren går gjennom kildekoden flere ganger før den kan levere resultatet. De forskjellige gjennomgangene gjør kontroller og bygger opp infrastruktur foran den påfølgende gjennomgangen inntil den siste genererer C++-kode. For brukeren er det viktig å vite hvordan dette virker inn på koden. Som sagt, prøver kompilatoren å gjette typene til funksjonsparametre, returtyper og variable, men dette gjøres på en ganske primitiv måte, og kun *float* og NumPy-array prøves.

4.4.1 Parsering

Koden sendes aller først til funksjonen *parse*, som følger med *compiler*-pakken, som er en del av den offisielle Python-distribusjonen. Denne funksjonen returnerer et abstrakt syntakstre eller slenger en *SyntaxError*. Dette syntakstreet er hva resten av gjennomgangene opererer med. De aller fleste gjennomganger gjør også modifikasjoner på objektene i syntakstreet.

Ettersom kompilatoren benytter den innebygde metoden for parsering, vil parseringen alltid støtte den samme versjonen av Python som brukes til kompilering. Kompilatoren er skrevet for versjon 2.5, men for enkelte endringer i språket, vil det være trivielt å oppdatere kompilatoren til senere Python-versjoner. Til gjengjeld er det ikke like trivielt å utvide Pythons syntaks for å støtte praktiske ting, slik Pyrex gjør. Men parserens kildekode er skrevet i Python, og det er mulig å lese den.

Resultatet av parseringen, syntakstreet, er ytterligere forklart i A.

4.4.2 Bygging av symboltabell

Symboltabellen er sentral i enhver kompilator. Bygging av symboltabellen foregår i to operasjoner. Først ser kompilatoren etter «deklarasjoner», det vil si setninger som definerer navn. Det vil for det meste være *global*-setninger, ordinære tilordninger og funksjons-

[1] *EMC* står for extension module compiler
[2] Engelsk: *namespace*

definisjoner. Disse navnene blir dyttet inn i symboltabellen, og hvert symbol får sin unike post, slik at en lokal variabel i én funksjon og en lokal variabel i en annen funksjon ikke refererer til samme post, selv om de skulle ha samme navn.

Etter at alle symboler er satt inn i symboltabellen, blir alle navneforekomster lenket til sin post i symboltabellen. I tillegg er *import*-setninger nå utført, slik at man kan bruke navnene de definerer. Hvordan man kan lage sine egne moduler, tar jeg opp under 4.6.1

Symboltabellen bygges automatisk av *Pyc.__init__*.

4.4.3 Deklarering av navn

Deklarering av navn er egentlig ikke det. Etter at *Pyc.__init__* er utført, ligger alle navn i symboltabellen, men med unntak av navn fra importerte moduler, er ikke navnene forbundet med noen type. Deklarering av navn er da ikke annet enn å definere objekters type. Det viktigste er i så måte argumenttyper, men alle navn kan deklareres. Udeklarerte navn blir senere forsøkt slått opp. Strategien er enkel, men jeg går gjennom den under 4.5.1.

Vanlige variable

Funksjonsargumenter og vanlige variable deklareres med *Pyc.typedef()*. Denne tar to argumenter: Variabelens navn og dens type. Navnet gis gjerne ved å angi *funksjonsnavn.variabelnavn*. F.eks. i

```
def foo(a):
    b = a + 42
    print b
```

kan man kompilere ved å si:

```
from pyemc.pyc import Pyc
comp = Pyc(code, mod)
comp.typedef('foo.a', int)
comp.compile()
```

I dette eksemplet kunne man også deklarert *b* ved å referere til *foo.b*.

Returverdi

Å sette en returverdi hos en kompilert funksjon er omtrent som å spesifisere verdien til en variabel. Denne gangen er navnet gitt ved funksjonsnavnet. Hvis man ville, kunne linjen

```
comp.typedef('foo', int)
```

eksekvert *før* comp.compile() sørget for at *foo* skulle returnere en *int* i C++. Ettersom det ikke finnes noen *return*-setning i *foo*, ville dette laget potensielt uhyggelige resultater, men kompilatoren vil anta at brukeren vet hva han gjør.

En interessant ting: I Python returnerer alle funksjoner en verdi. Hvis man ikke angir noen, returneres objektet *None*. En returverdi, enten *None* eksplisitt, eller implisitt, vil få kompilatoren til å generere en *void*-funksjon. Dette er litt annerledes fra hvordan det virker i Python, men et bevisst valg.

Funksjoner

Ettersom C++-kode genereres og kompileres, er det åpenbart at man bør kunne kalle bibliotekfunksjoner tilgjengelig for C++, men ikke nødvendigvis for Python. For eksempel kan man ha glede av de mange C-bibliotekene som er tilgjengelig på systemet.

For å bruke en slik funksjon, må den deklareres på forhånd. Kompilatoren trenger følgende: Hvilket navn funksjonen kalles ved i Python-kode, hvilket navn den kalles ved i C++-kode, navn på fil som deklarerer funksjonen i C++, hvilke argumenter funksjonen tar, og funksjonens returtype.

La oss ta et eksempel:

```
def foo(a, b):
    return bar(a) + baz(b)
```

Her er et eksempel på hvordan denne kan kompileres:

```
from pyemc.pyc import Pyc
comp = Pyc(code, 'mod')
comp.declare('bar', 'std::sin', float, float)
comp.declare('baz', 'std::tan', float, float)
comp.include('<cmath>')
comp.compile()
import mod
print mod.foo(3.14, 1.5)
```

Vi ser umiddelbart at funksjonene som deklareres kan gi totalt forskjellige navn i Python og C++. Hvis man skal støtte namespaces i C++, finnes det uansett ingen vei utenom å skrive om navnene.

Når det gjelder selve *Pyc.declare*, tar den argumentene Python-navn, C++-navn og returtype. Argumenter utover dette definerer argumenttypene til funksjonen som deklareres. Så i deklarasjonene er den første *float* returtypen, og den neste er den første – og siste – argumenttypen.

En annen fin ting er at kompilatoren støtter overlasting av funksjoner, det vil si at flere funksjoner kan ha samme navn i Python. Kompilatoren vil velge en funksjon basert på antall argumenter og argumenttypene. Dette fungerer relativt likt som i C++. Det er heller ikke nødvendig at de forskjellige funksjonene deler navn i C++. Grunnen bak dette valget, er at Python *egentlig* støtter overlasting – en funksjon kan simpelthen deklareres til å ta variabelt antall argumenter og bestemme sin oppførsel etter argumentenes antall og typer. Pythons innebygde løsning er i teorien mere fleksibel, men overlasting demmer opp for de fleste behov. Blant overlastede funksjoner velges den beste på denne måten:

- En passende funksjon må ha samme antall parametre som funksjonskallets argumenter.
- Hvert argument må kunne konverteres til den gitte parametertypen.
- Hvis det etter de foregående punkter står igjen flere alternativer, prøver kompilatoren å finne den funksjon hvis parametertyper står nærmest argumenttypene. Definisjonen av nærmest overlates til typebeskrivelsesklassene for typene.

4.4.4 Kodegenerering

Som leseren sikkert har lagt merke til, har *Pyc* en metode kalt *compile*. Denne kaller i sin tur *Pyc.inferTypes()* og *Pyc.codeGen()* hvis de ikke allerede er kalt. Disse to funksjonene står bak typededuksjonssystemet og generering av C++-kode. Etter dette vil *compile* håndtere C++-kompilatoren.

Typededuksjonssystemet bruker all typeinformasjon den har, og hvis den ikke har typeinformasjon angående argumenter, prøver den å gjette argumenttyper. Reglene for dette er nevnt, men her er de igjen:

- Alle argumenter antas først å ha typen *float*.
- Hvis de er forsøkt indeksert, er typen *ndarray* med datatypen C++-*double*. Den høyeste indeksen funnet avgjør hvor mange dimensjoner arrayen antas å holde.

Etter at typededuksjonen er ferdigstillet, kan kode genereres. Kodegenereringsfunksjonen lager to separate enheter. Den ene inneholder funksjonsinnmaten, mens den andre inneholder kode som integrerer C++ og Python. Kodegenerering utføres av *Pyc.codeGen()*.

Når koden er ferdig generert, holder kompilatorklassen koden. *Pyc.compile()* kan brukes. Denne skriver koden til fil, og kjører C++-kompilatoren. Går alt bra, kan man importere den ferdige modulen straks.

4.5 Typer

Den største forskjellen mellom C++ og Python er hvordan typesystemet fungerer. C++ er et *statisk* typet språk, noe som betyr at typene følger symbolene. Python er *dynamisk* typet, noe som innebærer at typene ikke følger symbolene, men *verdiene*. Dette gjør det helt nødvendig at en Python–C++-kompilator må ha informasjon om hvilke typer som brukes. Dette gjøres ved at kompilatoren forventer eksistens av diverse *typebeskrivere*[3], og for å gjøre forvirringen fullkommen, defineres dette systemet i modulen *pyemc.typewrapper*.

Typebeskrivere er klasser som implementerer en spesiell protokoll. Dette er diverse medlemsfunksjoner som kalles når operasjoner utføres for typen. En operasjon kan f.eks. være å bruke en operator med typen som et av argumentene, eller å kalle en medlemsfunksjon. Typebeskrivere må også kunne svare på om typen kan konverteres til en annen. I tillegg bærer typebeskriveren informasjon om hvilken type som tilsvares i C++.

[3] I C++ kalles slikt gjerne type traits.

Tabell 4.1: Typer støttet av kompilatoren

Type i Python	Typebeskriver	typewrap	C++-type
bool	pytype(bool)	Ja	bool
int	pytype(int)	Ja	int
float	pytype(float)	Ja	double
NoneType	pytype(None)	Ja	void/Python::NoneType
Ellipsis	pytype(Ellipsis)	Ja	Python::Ellipsis
long	pytype(long)	Ja	double
complex	pytype(complex)	Ja	std::complex<double>
C++-funksjoner	cfunc	Nei	
Overlastet funksjon	cfuncOverload	Nei	
str	Str	Nei	std::string
unicode	Unicode	Nei	std::wstring
tuple	Tuple(*typer*)	Nei	boost::tuples::tuple
list	List(*type*)	Nei	std::vector

Observante lesere har sett at eksemplene har benyttet typen *float*. Dette fungerer ved at *Pyc* benytter seg av objektet *typewrap* som ligger i *pyemc.typewrapper*. Dette objektet kan kalles, og bærer informasjon om en del typer, deriblant de fleste innebygde typene. Så kallet pyemc.typewrapper.typewrap(float) returnerer et typebeskriverobjekt for typen float.

Ikke alle typer kan slås opp med *typewrap*. For eksempel må lister og tupler ha informasjon om hva de skal kunne holde. På grunn av dette, holder *pyemc.typewrapper* informasjon om de innebygde typene, og typebeskriverne deres er klasser i denne modulen.

4.5.1 Typededuksjonssystemet

I avsnitt 2.2.3 viste jeg til at kompileringsprosessen for et kort program i Shed Skin var ganske tidkrevende. Dette skyldes Shed Skins typededuksjonssystem, som er relativt komplisert, og for at typededuksjonssystemet skal fungere, må Shed Skin analysere hele programmet. Å analysere hele programmet er uaktuelt for denne kompilatoren, og typededuksjonssystemet er mye enklere.

Systemet baserer seg på at en variabel er blitt tilordnet før den brukes. Dette er virker selvsagt i Python, men før betyr her at kompilatoren må ha sett tilordningen før den ser bruken. Det er mulig å koke sammen lovlige eksempler i Python hvor slikt ikke er tilfelle (se figur 4.1), men det har jeg antatt tilhører sjeldenhetene.

Typesystemet baserer seg på at alle uttrykk har en type. De minste byggestenene i typesystemet er såkalte *atoms*, eller atomer. Atomer kan være navn eller literale verdier. Et navn refererer da til et allerede definert objekt, mens literale verdier er heltall, flyttall, strenger og annet som kan tilordnes verdier. I AST'en representeres disse med objekter av typene *Name* og *Const*. I tillegg til dette, kan atomer være sammensatte objekter, som tupler (*Tuple*), lister (*List*) og mappinger (*Dict*), i tillegg til backtick-operatoren

Figur 4.1: Funksjon som tilordner variabel leksikalsk etter bruk

```
def foo():
    for i in range(10):
        if i > 0:
            print a
        else:
            a = 42
```

Variabelen a vil her være initialisert før bruk i Python, men siden a gis en type *etter* at den er blitt brukt i et uttrykk, vil kompilatoren klage. Dette kan her løses ved å deklarere a manuelt.

(*Backquote*). Generator-uttrykk og list comprehensions er ikke støttet av kompilatoren.

For å sette sammen atomer til uttrykk, bruker man operasjoner. Alle disse operasjoner kan oppfattes som funksjoner av en eller flere variable. For eksempel er f(x, y, z) en funksjon av variablene f, x, y og z, og kjenner man disse typene, har kompilatoren nok informasjon til å avgjøre typen til hele uttrykket. Operatorer finnes det mange av, og x + y kan ses på som en funksjon, f(x, y).

Når alle argumenttypene er kjent, kan kompilatoren evaluere funksjonens type, og finne rett overlasting basert på dette, og når rett funksjon er funnet, er uttrykkets type lik funksjonens returtype.

Når en variabel tilordnes, går kompilatoren først gjennom uttrykket variabelen settes til. Hvis variabelen ikke allerede er forbundet med noen type, deklareres variabelen der. På grunn av dette, bør man være nøyaktig når man initialiserer. 0 og 0.0 er to forskjellige typer, og å initialisere en variabel med 0 istedenfor 0.0 medfører at variabelen får typen *int* hvis den ikke deklareres manuelt.

4.5.2 Definisjon av nye typer

Enten man bruker kompilatoren til en enkelt operasjon eller lager egne moduler for kompilatoren, kan man definere sine egne typer. Dette er ikke vanskelig, men det kan være tidkrevende.

Normalt vil ikke typer ha sin egen plass i symboltabellen. Det er derfor to måter å opprette variabler av typen i kompilatoren: Enten ved å deklarere en variabel – f.eks. et argument – til å være av denne typen, eller ved å deklarere en funksjon som returnerer den aktuelle typen. Ettersom både C++ og Python betrakter et kall på selve typen som en konstruktør, er det en idé å deklarere konstruktøren som en funksjon som returnerer typen.

Når man skal definere en type for kompilatoren, må man implementere en typebeskriver. Dette er en klasse med en del medlemsfunksjoner definert. Jeg skal i resten av dette avsnittet anta at x og y er objekter av slike typebeskrivere.

Fra/til Python

To viktige funksjoner er x.extract(exp) og x.toPython(exp). Disse funksjonene kalles begge med C++-uttrykk i form av strenger som argumenter, og forventes å returnere nye C++-uttrykk basert på dette. Den første, *extract* mottar et argument av typen *PyObject**, og skal generere et uttrykk av den aktuelle typen, mens *toPython* mottar et uttrykk av den aktuelle types type, og skal avlevere et *PyObject**-objekt. Disse funksjonene kalles når Python sender med argumenter av typen, eller når typen returneres. Python-kompilatoren vil ikke generere kall til *Py_INCREF* eller *Py_DECREF*, så referansetelling er opp til disse funksjonene.

Typekonvertering

Funksjonen x.convertsTo(y) returnerer et positivt heltall hvis uttrykk av x' type kan konverteres til uttrykk av ys type. Hvis ikke bør den returnere y.constructFrom(x), som returnerer et positivt heltall hvis uttrykk av ys type kan konstrueres fra et uttrykk av x' type. Ellers returnerer den null. Ideen bak dette er at kompilatoren da kan vurdere funksjoner hvor det ikke er eksakt match mellom typene. Er returverdien positiv, må det etter C++' regler finnes en implisitt konvertering mellom typene. Størrelsen på returverdien skal gi et hint om hvor god konverteringen er. Kompilatoren vil foretrekke et lavt positivt tall foran et høyt.

Nødvendige strenger

Det er nødvendig for kompilatoren å kjenne typenes benevnelser i C++, men også benevnelser i Python er fornuftig. Kompilatoren vil kalle str(x) når x skal være med i en tilbakemelding til brukeren.

For å deklarere navn av typen, brukes funksjonen *regular()*. x.regular(s) kalles med strengen s for å deklarere en variabel ved navnet s inneholder. Kompilatoren sørger selv for å legge på semikolon. Sammen med *regular()*, har klassen *argument()*, som deklarerer typen som et argument. Den tar også navnet på variabelen som parameter. Tanken er at *argument()* kan deklarere variabelen som en referanse.

I tillegg til dette har typebeskriveren typens navn i C++ ved x.ctypename().

Operatorer

Operatorer behandles spesielt ettersom de behandles spesielt i C++ og Python. I Python kan brukeren implementere en operator ved å implementere en medlemsfunksjon med et spesielt navn. For eksempel betyr a+b det samme som den første av a.__add__(b) og b.__radd__(a). I C++ betyr det samme uttrykket enten operator+(a, b) eller a.operator+(b), men skjønt a.operator(b) ligner a.__add__(b), er det operator+(a, b) som blir foretrukket i C++.

For hver operator er det to par medlemmer typebeskriveren kan implementere. Det første brukes av typededuksjonssystemet, mens det annen brukes av kodegeneratoren. For addisjonsoperatoren er det første paret x.addType(y) og y.raddType(x). Det er

altså samme navn som funksjonsnavnet Python leter etter, men understrekene er fjernet, og *Type* er lagt til. x.addType(y) forventes å returnere en passende typebeskriver eller y.addType(x). Hvis operasjonen ikke har noen mening, kan operatoren slenge et unntak[4].

Paret som brukes av kodegeneratoren heter x.add(xepr, y, yexpr) og y.radd(yexpr, x, xexpr). Her er *xexpr* og *yexpr* strenger av typene. Disse er forventet å returnere en passende streng, som kompilatoren bruker direkte som C++-uttrykk. I motsetning til typefunksjonene er det ikke nødvendig å implementere disse funksjonene; kompilatoren vil selv anta at det er samme operator som brukes i C++ hvis funksjonen ikke finnes.

Medlemmer

Medlemmer implementeres vha. funksjonene x.attrType(name), hvor argumentet er en streng, og x.getattr(expr, name), hvor det første argumentet er en streng med C++-uttrykket for objektet. Som for operatorer forventes typefunksjonen å returnere en type, mens x.getattr(expr, name) forventes å returnere et C++-uttrykk.

4.6 Moduler

Moduler er støttet av kompilatoren. Når kompilatoren ser **import** foo, vil kompilatoren lete etter modulen *pyemc.lib.foo*. Finner den denne modulen, vil den prøve å la den laste inn informasjon ved å kalle *pyemc.lib.foo.load()* med to argumenter: Det første er et modulobjekt som kompilatoren bruker til å lete opp navn i modulen, og det annet er symboltabellobjektet. Denne funksjonen vil da laste opp alle navn, sørge for at riktige filer inkluderes i C++, og eventuelt registrere nye typer i *typewrapper.typewrap*. I tillegg vil gjerne typebeskrivere forbundet med modulen være tilgjengelig for brukere av kompilatoren gjennom modulen *pyemc.lib.foo*.

For at brukeren lettere skal kunne bruke kompilatoren, lastes moduler inn under utførelsen av *Pyc.__init__*. Selv om modulen ikke er importert i koden, kan den brukes. For eksempel kan det være ønsket at en parameter skal ha en type som finnes i en bestemt modul. Isåfall må modulen lastes inn før kompilering. Dette gjøres ved å kalle *Pyc.loadmodule()*. For eksempel vil kallet comp.loadmodule('numpy') laste inn *numpy*.

4.6.1 Egne moduler

Modulene samarbeider med resten av systemet uten kontakt med *Pyc*-objekter, som utgjør kompilatorgrensesnittet. Hovedjobben for modulinitialisering gjøres av funksjonen *load()*, som ligger i modulen. Denne kalles av kompilatoren, og gir modulen tilgang til symboltabellen.

Moduler i Python introduserer gjerne klasser/typer og funksjoner. I Python er alt egentlig objekter, men i C++ er det ikke slik. Derfor er det noen forandringer.

- Som tidligere vist, kan funksjoner overlastes. Det kan også være lurt å legge til overlastinger til innebygde funksjoner, som f.eks. *len()* og *str()*. I Python er det

[4] Dette skjer også automatisk om ikke funksjonen finnes i beskriverobjektet.

mange funksjoner som kaller definerte medlemsfunksjoner hos argumentene. I C++ er det vanligere å legge til overlastinger.

- I Python er typer objekter som kan kalles. Instanser av klasser skapes ved å kalle klassen selv[5]. Istedenfor å legge inn klasser som noe spesielt i symboltabellen, deklareres heller konstruktøren som en funksjon som returnerer typen.

- I tillegg kan klassene registreres hos *typewrap*. På denne måten kan moduler og brukeren enklere samarbeide.

Anta at et C/C++-bibliotek med funksjonene **double** func(**double** x) og **double** other(**int** y). Disse funksjonene deklareres i headerfilen *biblio.h*, som for demonstrasjonens skyld ligger i katalogen */some/path/include*, som ikke ligger i en av katalogene kompilatoren leter etter headerfiler i. Funksjonene er definert i biblioteket biblio, som ligger i */some/path/-lib/libbiblio.so*[6].

Det skal nå lages en modul slik at disse funksjonene kan brukes i pythonkompilatoren. Strategien er ganske rett frem:

- Man må legge inn et kompilatorflagg om at de aktuelle mappene skal søkes.

- Man må fortelle kompilatoren at den skal linke med *biblio*.

- Man må gi kompilatoren informasjon om at *biblio.h* må inkluderes.

- Man må fortelle kompilatoren informasjon om funksjonene.

Hvis denne modulen skal kalles *biblio*, må filen *pyemc/lib/biblio.py* opprettes. I denne filen skrives lastefunksjonen:

```
def load(mod, symtab):
    import pyemc.config, os.path
    # Compiler flags for dirs
    pyemc.config.comp_options.append('-I/some/path/include')
    # Link options for biblio
    pyemc.config.link_options.extend(['-L/some/path/lib', '-lbiblio'])
    # Include "biblio.h"
    symtab.addinc('"biblio.h"')

    import pyemc.typewrapper
    dbl = pyemc.typewrapper.typewrap(float)
    mod.declare('func', '::func', dbl, dbl)
    mod.declare('other,␣'::other',␣dbl,␣pyemc.typewrapper.typewrap(int))
```

Det første som gjøres her, er å modifisere objektene i *pyemc.config*. Det er disse som har flagg for bruk av kompilatoren. Akkurat som *Pyc*, har symboltabellobjektet en *addinc()*.

[5] Python er et dynamisk språk, og tillater andre måter, men dette er den absolutt vanligste måten.
[6] Dette er på et UNIX-system

Denne tar variabelt antall argumenter, og legger strengen til listen over filer som skal inkluderes. Kodegeneratoren skriver da en linje `#include fil`. Legg merke til at eller < og > må være med i denne strengen. For å deklarere funksjoner, brukes modulobjektets *declare*. Denne virker nesten som *Pyc*s funksjon for det samme, men den kaller ikke *pyemc.typewrapper.typewrap*.

4.7 Samhandling med SWIG

Det er to ting som kan gjøres med SWIG. For det første kan man bruke SWIG for å generere wrapperen som konverterer Python-argumenter til C++ og returverdien tilbake til Python. For eksempel kan det hende man allerede har skrevet en *typemap* for noen av typene. Det kan også hende man er interessert i å bruke SWIG-ede objekter. Eller kanskje man foretrekker SWIGs avanserte metoder.

4.7.1 Typemappere

SWIG baserer seg på typemappere for å konvertere Python-typer til C++-typer, for å konvertere C++-typer til Python-typer og til å sjekke. Python-kompilatoren bruker også typemappere. Typebeskriverne har, som beskrevet, medlemmene *toPython* og *extract*. Gitt en typebeskriver, x, kan man generere typemappere med følgende Python-kode:

```
inmap = """%%typemap␣(in)␣%s␣{
␣␣␣␣$1␣=␣%s;
}
""" % (x.ctypename(), x.extract('$input'))

outmap = """%%typemap␣(out)␣%s␣{
␣␣␣␣$result␣=␣%s;
}
""" % (x.ctypename(), x.toPython('$1'))
```

Man skal aldri utelukke at grunnen til å bruke SWIG er at man trenger flere eller mere avanserte typemappere enn kompilatoren tillater. Da vil isåfall ikke denne teknikken virke. Men på de typene som følger med kompilatoren, vil disse fungere.

4.7.2 SWIG-ede typer

En ting man kan gjøre med SWIG, er å generere språktilpasninger for biblioteker skrevet i C++. Har man en slik type i Python, er det en relativt grei sak å trekke den ut av en *PyObject** og tilbake igjen, og dette er lettest ved å bruke SWIG selv.

Kommandoen `swig -python -external-runtime fil` genererer en fil ved navn *fil* (som angitt på kommandolinjen). Denne filen inneholder SWIGs grensesnitt, som forøvrig genereres av SWIG til alle wrapperfiler. Om man enten skriver en typemap for bruk av SWIG, eller en typemap for bruk av kompilatoren, har man tilgang til det man trenger.

SWIG-ede typer kan ofte bruke følgende typemaps, på samme måte gitt vha. en typebeskriver x:

```
inmap = """%%typemap␣(in)␣%s␣{
␣␣␣␣%s␣*tmp␣=␣&$1;
␣␣␣␣if␣((SWIG_ConvertPtr($input,␣(void␣**)␣&tmp,␣$descriptor(%s␣*),␣0))␣==␣-1)␣return␣NU
}
""" % (x.ctypename(), x.ctypename(), x.ctypename())

outmap = """%%typemap␣(out)␣%s␣{
␣␣␣␣$result␣=␣SWIG_Python_NewPointerObj((void*)&$1,␣$descriptor(%s␣*),␣0);
}
""" % (x.ctypename(), x.ctypename())
```

Men når man først lager en typebeskriver for en SWIG-et klasse, kan man likegjerne sørge for at x.extract() og x.toPython() fungerer som de skal.

4.7.3 Exceptions

Python benytter seg av exceptions. C++ benytter seg av exceptions. Dessverre er de ikke like, og skulle en exception forsvinne inn i Pythons tolker, vil fæle ting skje. På side én i boken om C++-exceptions står det: «La aldri en exception falle ut i C-kode», og Python er skrevet i C.

Noen av funksjonene i kompilatoren kan slenge exceptions. Spesielt gjør konverteringene det. For at alt skal virke godt, bør derfor både typemaps og selve funksjonskallet pakkes inn i *try*. Man kan gjøre det i typemapene manuelt, og *%exception*-kommandoen i SWIG legger selve kallet inn i en *try*-blokk:

```
%exception {
        try {
                $action;
        }
        catch (Python::BaseException &e) {
                PyErr_SetString(e.pyexc(), e.what());
                return NULL;
        }
}
```

Leser man denne, legger man også merke til følgende: *Python::BaseException* er definert i *exceptions.hpp*, og medlemmet *pyexc()* returnerer faktisk et exception-objekt Python kan bruke.

4.7.4 Generering av SWIG-grensesnitt

Skjønt det ikke hadde vært spesielt vanskelig å lage, har ikke *Pyc* noe grensesnitt som genererer SWIG-filer. Men det er ikke altfor vanskelig å gjøre selv. For å generere en

grensesnittfil, trenger man tilgang på den genererte kode. Etter kodegenerering holder *Pyc*-objektet modulens C++-kildekode i attributtet *mod*. Legger man så på typemaps, så er man i havn.

5 Sammenligning

Et av målene var å få kjøretiden ned. I dette kapitlet vil jeg gjøre noen sammenligninger. Førsteprioritet for kompilatoren er numerisk kode, spesielt ved bruk av NumPy; derfor vil også numerisk kode med NumPy ta en stor del av dette kapittel.

Kompilatoren genererer C++-kode, og mange har tidligere testet C++-kode mot Python-kode. Det er velkjent at C++-kode kjører raskere enn Python, så like viktig som selve målingene er kvalitative vurderinger om hvordan C++-koden er oversatt.

5.1 Eksempel fra figur 1.1

Aller først eksemplet fra figur 1.1. Kompilatoren kompilerer Python-filen (jeg har fjernet kallet på *main()*) direkte, og resultatet ser slik ut:

```
namespace emceff {
    void f(double a)
    {
        ;
    }
    void main()
    {
        double a;
        double c;
        double b;
        int i;
        double t;
        t = Python::clock();
        i = 0;
        while ((i < 10000000)) {
            a = 5.0;
            b = 3.7;
            c = ( (a * b) * i) ;
            f(c);
            i += 1;
        }
        t = (Python::clock() - t) ;
        std::cout <<t<< std::endl;
    }
}
```

Den eneste forskjellen i løkken er her at variablene deklareres utenfor løkken. Noe annet er ikke mulig uten en mere komplisert bruksanalyse av variablene. Et lite men er også at variablene ikke initialiseres med deklarasjonen, noe som har større betydning for klassetyper. Problemet er at det er teoretisk vanskelig å bestemme initialiseringspunktet statisk, og jeg ønsker ikke å bruke pekere til variabler.

Funksjonen *Python::clock* er en wrapper rundt *std::clock*. Den konverterer kun resultatet til en *double*, slik *clock* virker i Python, og funksjonen må forventes å bli inlinet når det kompileres med optimalisering.

Ikke overraskende skriver denne funksjonen nøyaktig samme tid som C++-versjonen i eksemplet: 0,17.

En ting jeg ønsker å kommentere, dog: Kodeeksemplet har to flyttallsliteraler. Pythons parser transformerer disse til flyttallsverdier selv, og dette skjer før kompilatoren engang har fått fingrene i koden. Det er derfor mulig at flyttallsverdier endrer form fra Python til C++, og siden flyttallsverdiene ikke skrives med mange desimaler, kan det tenkes at de endrer verdi i helt spesielle tilfeller. Dette er selvsagt beklagelig, men skulle ikke være et stort problem. Ved å sende tallet som argument, kan man påse at alt går riktig for seg.

5.2 Daxpy

Daxpy er en operasjon, så vidt jeg vet, hentet fra biblioteket *BLAS*, et lineær algebrabibliotek. Det viktige er funksjonen: Daxpy kalkulerer

$$ax + y$$

for vektorer x og y og en skalar a.

En grunn til å teste akkurat denne, er at den har noen fine egenskaper. Den kan vektoriseres i NumPy. Hvis a er en *float*-variabel, og x og y er arrayer, vil uttrykket a∗x+y gi verdien til operasjonen. Det er samtidig enkelt å skrive en løkke som beregner uttrykket.

Daxpy-operasjonen er også kjent for alle som har sett på Blitz++-biblioteket. I [12] er akkurat denne brukt for sammenligning. Dette har med hvordan Blitz++ utfører binære operasjoner. a∗x + y kan deles opp i to: Først utføres a∗x, og resultatet lagres i en midlertidig variabel, som brukes av addisjonen. Også denne lager en ny variabel. Bruk av slike midlertidige variable kan være ekstremt ressurskrevende når objektene er store.

Alternativt til å bruke operatoroverlasting kan man skrive hele uttrykket til en løkke, og for daxpy er dette trivielt: **for** i **in** len(x): r[i] = a∗x[i] + y[i]. I ren Python er løkken mye tregere enn operatorene, som NumPy implementerer i C.

Hvilke kvaliteter har daxpy som gjør at Blitz++-forfatterne bruker nettopp denne som eksempel? Svaret er at Blitz++ bruker template-teknikker for å unngå temporære variable. Heller enn å returnere en ny array, vil de aritmetiske operatorene returnere template-objekter. Disse er utstyrt med indekseringsmuligheter. I Blitz++ vil uttrykket x+y returnere et objekt som kan minne noe om:

```
template <typename Lhs, typename Rhs>
class OpPlus {
```

```
    const Lhs &lhs;
    const Rhs &rhs;
    public:
    OpPlus(const Lhs &lhs, const Rhs &rhs) : lhs(lhs), rhs(rhs) {}
    double operator[](std::size_t i) const { return lhs[i] + rhs[i] }
};
```

Typeparametrene kan her være alt som kan indekseres, f.eks. array-objekter eller andre operatorer. Dermed kan bruk av uttrykket, f.eks. i en tilordning, lage en løkke og sette hvert element for seg. Ved bruk av kompilatorer som inliner små funksjoner, som denne indekseringsoperatoren, vil hele uttrykket virke like raskt som om man skrev løkkeformen selv. Ettersom kompilatoren benytter seg av wrappere skrevet i C++ for bruk av NumPy-arrayer, skulle det være mulig – uten å endre på selve kompilatoren – å benytte denne teknikken for operasjoner på NumPy-arrayene. Av tidshensyn er det dog ikke gjort.

En annen optimalisering ville være å gjenbruke temporære objekter. Hvis addisjonsoperatoren hadde visst at venstresiden var et midlertidig objekt som uansett skulle skrotes etter addisjonen, kunne den ha gjenbrukt dette objektet. Med NumPy ville det vært omtrent som å skrive om til

```
r = a*x
r += y
```

Her blir det bare skapt ett midlertidig objekt, og i denne situasjonen måtte man uansett hatt et sted å lagre resultatet av hele operasjonen. Skjønt det ville vært mulig for kompilatoren å vite at a∗x er et temporært objekt, og dermed kunne kompilert til å gjenbruke dette objekt, er dette vanskelig; Python har selv ikke støtte for dette, og å legge det inn ville ha vært relativt mye arbeid. Men en interessant sak er at C++[1] kommer til å få støtte for gjenbruk av temporære objekter, og spesialversonen for C++0x av GNU-kompilatoren har støtte for såkalte *rvalue references*. Kort betyr det at man kan skrive en spesiell overlasting som blir valgt hvis argumentet er en rvalue, altså ikke forbundet med noen variabel.

Listing 5.1: Daxpy i Python

```
def daxpy(a, x, y, r):
        t = time.clock()
        for i in range(len(r)):
                r[i] = a*x[i] + y[i]
        return time.clock() - t
```

Her vil x, y og r være NumPy-arrayer, og a et flyttall. Jeg har testet koden med kompilatoren min, med vanlig Python, og med Python og Psyco. Ettersom koden medfører bruk av NumPy-arrayer, kan den ikke kompileres med Shed Skin. Men først: Hva må man kunne forvente? Av koden vil jeg tro C++-versjonen er raskest, og av resultatet av koden i figur 1.1, vil jeg forvente 100 ganger fartsøkning i forhold til Python også denne gang. Denne koden må også karakteriseres som «algoritmisk kode», og Psyco gjør gjerne

[1] Neste versjon av standarden kommer rundt 2009

en god jobb med slik kode. Det er et problem: NumPy-arrayer er ikke en «kjent» type. Den store kveleren for Psyco her vil være indekseringsoperatoren. Den er skrevet i C, og hver indeksering krever da at Psyco lager et kall til objektets indekseringsfunksjon. Likevel bør Psyco levere kjappere kode enn ren Python.

La oss først se på den genererte koden.

Listing 5.2: Daxpy i C++

```
namespace daxpy_cc {
   double daxpy(double a,
    Python::numpy::ndarray<double, 1> x,
    Python::numpy::ndarray<double, 1> y,
    Python::numpy::ndarray<double, 1> r)
   {
      int i;
      double t;
      t = Python::clock();
      for ( i = 0 ;  i < Python::len(r) ;  i += 1 ) {
         r(i) = ( (a * x(i))  + y(i)) ;
      }
      return (Python::clock() - t) ;
   }
}
```

C++-versjonen synes visuelt veldig lik Python-versjonen. Med unntak av mange parenteser[2] og at *range* er erstattet med en indeksvariabel[3], er det praktisk talt identisk. En forskjell er at parametrene er verdi-parametre, men array-objektene er i C++ implementert med pekersemantikk[4].

Denne funksjonen har jeg så kalt med arrayer med en million elementer. Dette har jeg gjort 300 ganger med C++-versjonen, 10 ganger i Python, og 30 ganger med Psyco. Summen av returverdiene har jeg så justert for forskjellig antall kall. Resultatet er presentert i tabell 5.1.

Disse tidene er, som leseren kanskje har innsett, målt bare over løkken. Tid Python bruker til kall på funksjonen og på å gjøre dette i en løkke, samt tid brukt for å konvertere verdiene fra Python til C++, er da ikke med i målingene.

Ingen vettug person ville skrevet noe slikt med NumPy, dog. NumPys styrke ligger i de vektoriserte funksjonene, og i de tilfellene man *kan* bruke disse, gjør man det; det er dessverre ikke alltid man kan vektorisere. Istedenfor løkker og mange kall på indekseringsfunksjonen, kan man skrive vektorisert kode. Dette ser man eksempel på i følgende eksempel:

[2] Dette skyldes at det er lettere å generere parenteser hver gang enn å sjekke om de er nødvendige.
[3] Psyco gjør det samme
[4] Eller på norsk: objektene inneholder en peker til selve arrayene, og kopiering av objektet innebærer kopiering av denne pekeren.

Tabell 5.1: Tider skrevet ut av daxpy-programmet i listing 5.5

Time(C++):	8.4
Vect(C++):	25.6
Time(Python):	1017.0
Vect(Python):	20.8
Time(Psyco):	775.4
Vect(Psyco):	19.4

Listing 5.3: Vektorisert daxpy

```
def vdaxpy(a, x, y):
    t = time.clock()
    r = a*x + y
    return time.clock() - t
```

Denne kan også oversettes til C++, og resultatet ser man her:

Listing 5.4: Listing 5.3 kompilert til C++

```
namespace daxpy_cc {
   double vdaxpy(double a,
    Python::numpy::ndarray<double, 1> x,
    Python::numpy::ndarray<double, 1> y)
   {
      Python::numpy::ndarray<double, 1> r;
      double t;
      t = Python::clock();
      r = ((a * x)+y);
      return (Python::clock() - t) ;
   }
}
```

I sentrum står akkurat samme uttrykk som i Python, så dette måler vel egentlig ikke annet enn hvor kjappe de vektoriserte funksjonene er. Kjøring av dette eksemplet er også fremstilt i tabell 5.1.

Av kjøretidene i 5.1, kan vi lese at det lønner seg å oversette koden til C++. I dette eksemplet er koden dobbelt så rask som det vektoriserte alternativet i Python. C++-biblioteket er til gjengjeld ikke like raskt som NumPys vektoriserte alternativer, men i de tilfeller man kan bruke vektoriserte NumPy-funksjoner, er man uansett kommet langt på vei.

Listing 5.5: Tidsmåler for daxpy

```
import os.path

if not os.path.exists("daxpy_cc.so"):
    from pyemc.pyc import Pyc
    comp = Pyc("""
import␣numpy,␣time
def␣daxpy(a,␣x,␣y,␣r):
␣␣␣␣t␣=␣time.clock()
␣␣␣␣for␣i␣in␣range(len(r)):
␣␣␣␣␣␣␣␣r[i]␣=␣a*x[i]␣+␣y[i]
␣␣␣␣return␣time.clock()␣−␣t

def␣vdaxpy(a,␣x,␣y):
␣␣␣␣t␣=␣time.clock()
␣␣␣␣r␣=␣a*x␣+␣y
␣␣␣␣return␣time.clock()␣−␣t
""", 'daxpy_cc')

    import pyemc.lib.numpy as np
    comp.typedef('vdaxpy.x', np.array())
    comp.typedef('vdaxpy.y', np.array())
    comp.compile()
    del comp

import daxpy_cc, numpy

a = numpy.ones((1000000,), dtype=float)
b = numpy.ones((1000000,), dtype=float)
r = numpy.ones((1000000,), dtype=float)

tim = 0.0
for i in xrange(300):
    tim += daxpy_cc.daxpy(42., a, b, r)

print "Time(C++):␣␣␣␣", tim

tim = 0.0
for i in xrange(300):
    tim += daxpy_cc.vdaxpy(42., a, b)

print "Vect(C++):␣␣␣␣", tim
```

```
import time
def daxpy(a, x, y, r):
    t = time.clock()
    for i in range(len(r)):
        r[i] = a*x[i] + y[i]
    return time.clock() - t

def vdaxpy(a, x, y):
    t = time.clock()
    r = a*x + y
    return time.clock() - t

tim = 0.0
for i in xrange(10):
    tim += daxpy(42., a, b, r)

print "Time(Python):␣", tim*30

tim = 0.0
for i in xrange(300):
    tim += vdaxpy(42., a, b)

print "Vect(Python):␣", tim

import psyco
psyco.full()

tim = 0.0
for i in xrange(30):
    tim += daxpy(42., a, b, r)
print "Time(Psyco):␣␣", tim * 10

tim = 0.0
for i in xrange(300):
    tim += vdaxpy(42., a, b)

print "Vect(Psyco):␣␣", tim
```

6 Avslutning

6.1 Vurdering av prosjektet

Dette prosjektet har vært å implementere en Python-kompilator. I oppgaven har jeg vist at den virker, og etter oppgavens mål kompilerer den Python-kode til meget effektiv C++-kode. Det er selvfølgelig noen negative sider.

Kompilatoren er ikke ferdig. Det hadde også vært for mye å forlange at hele kompilatoren skulle være ferdig i løpet av én masteroppgave. Python er et forferdelig stort språk; riktignok er ikke syntaksen avskrekkende, men språket består i stor grad av forskjellige objekter også. Men det viktigste av rammeverket er på plass, og mye av det som gjenstår er å implementere standardbibliotek.

En litt subjektiv vurdering er at kompilatoren kompilerer C++ skrevet med Python-syntaks til C++. De største forskjellene mellom C++ og Python i mine øyne er at C++ er statisk typet mens Python er dynamisk typet, samt at det interaktive Python-skallet gjør det lett å teste saker og ting i Python, eller med andre ord: Det er fort gjort å komme igang med mindre ting i Python. Som diskutert tidligere i denne oppgaven er det en nødvendighet at kompilatoren har flere krav enn Python har, og sammen med at det å kompilere er en omstendelig prosess, føles bruken av den som en dragning mot bruk av C++.

6.2 Oppsummering

Det finnes mange verktøyer som bøter på Pythons trege adferd, noe som klart demonstrerer at mange jobber med Python og ser at det noen ganger går for sakte. SWIG er et populært verktøy, og mange mindre verktøyer bygger på SWIG. Derfor er også SWIG interessant, og jeg har vært så vidt innom bruk av dette verktøyet.

Systemet virker, om enn noe omstendelig. En veldig omstendelig prosess er å skrive typebeskrivere. For denne kompilatorens del er typebeskrivere et nødvendig onde. Men et positivt produkt av typebeskriverne er typededuksjonssystemet – uten det hadde man måttet deklarere alle variable.

Kompilatoren lar brukeren definere egne moduler, med typer og funksjoner. Dette lar brukeren utvide kompilatorens reportoire, og samtidig kan man introdusere biblioteker som ikke er portet til Python.

Sammenlignet med andre alternativer fungerer kompilatoren meget godt. Kompilatoren kan også brukes med NumPy-kode, noe ikke alle alternativer kan.

Konklusjonen er at det lar seg gjøre å generere effektiv C++-kode fra Python-kode. Dette gjør kompilatoren til et alternativ når mindre deler av et Python-program skal

effektiviseres. Kompilatoren kan også brukes til å gjøre Python-kode lettere tilgjengelig fra C++.

6.3 Fremover

Kompilatoren kan forbedres. For det første må de bibliotekene som følger med, skrives ferdig. Dette vil medføre litt arbeide med å implementere deler funksjoner i C++, deklarere dem i modulene og skrive typebeskrivere.

Det å skrive typebeskrivere er en omstendelig prosess, som kunne vært forenklet ved å la Python generere typebeskriverne fra enklere og mere kortfattede beskrivelser.

Kompilatoren har nok også en del lus. Disse må lukes ut. Det ville også vært fornuftig å gjennomgå prosessene for funksjonskall på nytt. Kompilatoren kan også dra nytte av bedre feilmeldinger når noe går galt på grunn av programmeringsfeil i koden.

Det er relativt enkelt å koble den genererte koden sammen med SWIG. I tillegg bør man kunne koble kompilatoren sammen med verktøyer som Instant.

Bibliografi

[1] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP '95 – Object-Oriented Programming: 9th European Conference, Åarhus, Denmark, August 1995. Proceedings*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[2] Mark Dufour. Shed skin – an optimizing python-to-c++ compiler. Master's thesis, Delft University, 2006.

[3] Douglas Gregor, Jaakko Järvi, Jens Maurer, and Jason Merrill. Proposed wording for variadic templates. Technical report, JTC1/SC22/WG21 – The C++ Standards Committee, 2007.

[4] ISO/IEC, Geneva. *ISO/IEC 14882 – Programming languages – C++*, second edition, 2003.

[5] Donald E. Knuth. *The Art of Computer Programming*, volume 1 – Fundamental algorithms. Addison Wesley, third edition, 1997.

[6] Donald E. Knuth. *The Art of Computer Programming*, volume 3 – Sorting and Searching. Addison Wesley, second edition, 1998.

[7] Hans Petter Langtangen. *Python Scripting for Computational Science*. Springer-Verlag, second edition, 2005.

[8] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley Publishing Company, third edition, 1997.

[9] Herb Sutter. "export" restrictions, part 1. *C/C++ User's journal*, 20(9), September 2002.

[10] Herb Sutter. "export" restrictions, part 2. *C/C++ User's journal*, 20(11), November 2002.

[11] The Unicode Consortium. *The Unicode Standard, Version 5.0*, 2006.

[12] T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

A Syntaks for compilers AST

Pythons compiler.parse() returnerer et abstrakt syntakstre (AST). Dette er til tider dårlig dokumentert. Derfor vil jeg her gi en beskrivelse av dette. Dokumentasjonen i dette kapittel er i stor grad basert på kildekoden som følger med Python.

Pakken compiler.ast inneholder en mengde klasser som beskriver et abstrakt syntakstre for Python. Selve pakken er i seg selv generert under kompilering av Python. Det samme er dens dokumentasjon.

Jeg vil her presentere en EBNF-lignende syntaks for disse AST-objektene. Syntaksen presenterer hva det er mulig å få ut av compiler.parse(), og har vært nødvendig for meg under implementeringen. Syntaksen har følgende form: En linje som inneholder «:=» vil si noe om en klasse og dens medlemmer. Klassenavnet står på venstre side av «:=», og attributter objekter av klassen har, står på høyre side. Disse er adskilt av komma, og hvert attributt er på formen «navn=element». Element kan være etterfulgt av «?», «+» eller «*», som betyr henholdsvis: attributtet kan være *None*, attributtet er en liste med minst ett element og attributtet er en liste, men ikke nødvendigvis noe innhold. Elementet sier ellers hva slags objekter som kan lagres i attributtet. Det kan være et AST-klassenavn, et pseudonavn, et gruppenavn eller en liste med klasse- og gruppenavn adskilt med «|» og plassert i parentes. Gruppenavn eller liste betyr at én av klassene denne konstruksjonen nevner direkte eller indirekte kan lagres. I tillegg har noen av objektene et attributt, *flags*, hvis verdi er angitt som et argument andre steder. Et pseudonavn betyr at attributtet har en annen verdi, for eksempel en streng. Grupper defineres ved å bruke «=». Dette definerer navnet på venstre som en gruppering av alle navnene på høyre side. Likhetstegnet brukes også for å definere pseudonavn.

A.1 Module

```
Module := doc=Doc?, node=Stmt
Doc = doc string
```

Klassen Module beskriver en modul. En modul er da en kompileringsenhet, og compiler.parse() vil alltid returnere et Module-objekt.

Et Module-objekt har to medlemmer: doc og node. Doc er modulens doc-string eller None. Node er et Stmt-objekt som inneholder modulens kode.

A.1.1 Stmt og setningsobjekter

```
Stmt := nodes=Statement*
```

Stmt representerer ikke en setning, men en gruppe setinger på samme nivå. Enkelt kan man si at disse setningene har til felles at de har samme innrykk og står rett ved

hverandre. I pythons grammatikk, representerer Stmt en suite, og Stmt-objekter vil alltid være barn av Module, Function, For, While, If og alt som underordner en eller flere setningsgrupper. Et Stmt-objekt har kun ett medlem, nemlig nodes, som er en list av statement-objekter.

Det er også slik at resten av AST-klassene kan deles i to grupper: Setninger og resten. Setninger returnerer aldri en verdi, og forekommer også bare som barn av Stmt. Resten representerer uttrykk, men det finnes begrensninger og oppdelinger innenfor disse.

Disse objektene regner jeg som Statement-objekter:

```
Statement =
  Assert |
  Assign |
  AugAssign |
  Break |
  Class |
  Continue |
  Del |
  Discard |
  Exec |
  For |
  From |
  Function |
  Global |
  If |
  Import |
  Pass |
  Print |
  Printnl |
  Raise |
  Return |
  TryExcept |
  TryFinally |
  While |
  With |
  Yield
```

Assert

```
Assert := test=Exp, fail=Message
Message = error message string
```

Assert representerer en assert-setning. En assert-setning ligner på et kall på assert() i C++, men er ikke helt den samme. I C++ brukes et assertkall typisk til å si: «Jeg, programmereren, forventer at dette skal holde, men hvis det er feil i programmet, gjør det ikke det.» I C++ er assert() en makro, og den gjør intet hvis makroen NDEBUG er

definert.

I Python er assert en egen setning. Kalles Python med opsjon for optimalisering, vil assert ikke eksekveres. Jeg anser det derfor fornuftig å oversette assert med assert. Python tilbyr ekstra funksjonalitet ved assert: Man kan legge ved en feilmelding. Denne informasjonen vil forsvinne i kompileringen.

Assert-objektet har medlemmene test og fail, som er hhv. AST til et boolsk uttrykk og None eller et uttrykk som gir feilmeldingen.

Assign

```
Assign := nodes=AssLvalue(flags=OP_ASSIGN)+, expr=Exp
```

Assign er tilordning ved bruk av operatoren «=». I Python er Assign også en deklarasjon. Et navn som står på venstre side av tilordningsoperatoren blir deklarert med tilbakevirkende kraft i den aktuelle lokale konteksten. Jeg har valgt å legge inn regelen om at bare disse navnene kan er lokale variable i en funksjon, selv om man i Python har mulighet til å manipulere symboltabellen.

Assign tillater, som i C++, at man kjeder sammen flere venstresider. Assign har derfor følgende medlemmer: nodes og expr, som er hhv. en liste med en eller flere venstresider, og et uttrykk. Venstresidene kan hentes fra et veldig begrenset utvalg, og jeg kommer til definisjonen av AssLvalue senere.

AugAssign

```
AugAssign := node=(Name|Slice|Subscript|Getattr), op=OpAugAssign, expr=Exp
OpAugAssign = augassign operator
```

AugAssign er kortformene «+=», «-=» osv. Den største forskjellen mellom AugAssign og Assign er at AugAssign ikke vil deklarere noen variable. AugAssign kan heller ikke kjedes sammen. Man kan også legge merke til at venstresiden i AugAssign er noe annerledes fra den i Assig.

AugAssign har tre medlemmer: venstresiden, node; operatoren, op; og høyresiden, expr. Venstresiden gis ved et navn, et uttrykk på formen «uttrykk[...]» eller «uttrykk.navn». Operatoren gis som en streng, og høyresiden er et uttrykk.

Break

Break har ingen medlemmer. Dersom løkker er oversatt til løkker, og ingenting annet er oversatt til løkker eller switch-setninger, kan break oversettes med break.

Continue

Continue har en annen semantikk, men brukes akkurat som break.

Class

```
Class := name=Id, bases=Name+, doc=Doc, code=Stmt
```

Klasser kan nok være praktisk, men for øyeblikket har jeg valg å ikke ta dem med. Klasser vil ganske enkelt utgjøre for mye jobb.

Del

```
Del = AssLvalue(flags=OP_DELETE)
```

Del kan anta flere former. Den kan slette elementer fra lister og dicter. Den kan også slette navn fra symboltabeller. Det siste er ikke forenlig med C++, og kan følgelig ikke implementeres. Det første kan ofte materialiseres til å kalle en slettefunksjon i et objekt.

Discard

```
Discard := expr=Exp
```

Discard representerer expression statements, eller setninger som bare består av et uttrykk, for eksempel funksjonskall. I C++ er de fleste setninger akkurat dette, så Discard vil ikke trenge å gjøre mer enn å sette et semikolon bak uttrykket.

Exec

```
Exec := expr=Exp, locals=Exp, globals=Exp
```

Exec er i seg selv hva som skiller et scriptesprog fra et kompilert sprog. Exec kaller vilkårlig Python-kode, noe som i seg selv bør være overkommelig (ved å kalle Pythontolkeren). Men C++ kan ikke sende med symboltabellen som en dict, og dette er grunn god nok til ikke å bry seg om exec for denne gang.

For

```
For := assign=AssLvalue(flags=OP_ASSIGN), list=Exp, body=Stmt, else_=(Stmt|None)
```

For-løkken i Python er noe forskjellig fra for-løkken i C++. Det idiomatiske **for** i **in** range(\ldots) behandles spesielt, mens annen bruk av for-løkke oversettes med en løkke som bruker iterator.

From

```
From := modname=Id, names=Id+
Id = identifier
```

From er en variant av Import. Istedenfor å deklarere modulnavnet, henter den inn navn fra andre moduler. For å kompilere from, er det likevel nødvendig å importere på vanlig måte først, men da uten å binde modulnavnet eller dets alias.

Function

```
Function := decorators=(None|Decorators),
  name=Id, argnames=Id*, defaults=Exp*, flags,
  doc=Doc, code=Stmt
```

Function definerer funksjoner.

Global

```
Global := names=Id+
```

Global deklarerer navn til å være ikke-lokale. Disse variablene må sendes med på annen måte. Navnene sendes med som en liste med strenger i medlemmet names. Disse vil da ikke bindes og defineres av Assign.

If

```
If := tests=Test+, else_=(None|Stmt)
Test = [0]=Exp, [1]=Stmt
```

If-setningen matcher godt sin C++-ekvivalent. If-objekter representerer en kjede if-elif-else. Medlemmet tests er en liste med totupler, som består av AST for testuttrykket og det tilhørende Stmt-objektet. I tillegg er else_ en siste mulighet, og hvis den er satt, representerer denne else-delen. Ikke overraskende vil jeg oversette if med C++' ekvivalent. Den eneste forskjellen er at det i C++ heter else if, men dette løses ved å være påpasselig med krøllparenteser.

Import

```
Import := names+ names = [0]=Id, [1]=(None|Id)
```

Import deklarerer moduler. Modulene bindes mot sitt eget navn hvis names[n][1] er en streng, ellers mot det angitte navn. Import-setningen kan også navngi nye pakker. Dette gjøres ved at names[n][0] inneholder ett eller flere punktumer. Isåfall står modulnavnet etter det siste punktumet.

Pass

Pass er Pythons versjon av null-setningen. Den brukes når syntaktiske krav krever en setning. C++ har også sånne krav, men de kan alltid løses opp ved bruk av krøllparenteser. Ellers tillater C++ også at setningens uttrykk droppes, og at man bare står igjen med semikolon.

Print og Printnl

```
Printnl|Print := nodes=Exp+, dest=(Exp|None)
```

Print og printnl sørger for utskrift til fil eller stdout. Dest skal evalueres til et file-objekt. Hvis dest er None, sendes utskrift til stdout. Den enkleste måten å få dette til å fungere på, er å benytte seg av ostream-objekter i C++. For øyeblikket tar jeg sikte på utskrift til stdout.

Raise

```
Raise := expr1=expr2=expr3=None
Raise := expr1=Exp, expr2=Exp, expr3=Exp
```

I Python heter det «Raise an exception». Dette er omtrent det samme som throw i C++.

Den første formen brukes i en Except-blokk, og lar den behandlede exception kastes på ny. Dette samsvarer godt med den tilsvarende throw uten argumenter i C++.

I Python er exceptions mer dynamiske enn i C++, og man har også en syntaks som minner mer om syntaktisk sukker. Raise har opptil tre argumenter. Den tredje kan være et Trace-object. Dette er isåfall hva som vises brukeren hvis programmet avsluttes. C++ har ingen ekvivalent for dette, så tredje argument ignoreres hvis satt.

Første argument kan være en type. Om så er, kan C++ støtte dette hvis det er et navn. Dette navn må kunne kalles med annet argument (er dette en tuppel, så kalles navnet med flere argumenter) som argument.

Dersom første argument ikke er en type, slenges første argument som seg selv.

Return

```
Return := value=(Exp|None)
```

I Python fungerer return omtrent som i C++. Ettersom Python-funksjoner alltid returnerer en verdi, må funksjonens returverdi deduseres fra alle returpunktene. I Python er returverdien None det samme som fraværet av returverdi.

TryExcept

```
TryExcept := body=Stmt, handlers=Handler+, else_=(Stmt|None)
```

Den største forskjellen mellom Pythons TryExcept, er else-blokken. Dette kan implementeres i C++ med goto. Ellers tror jeg at try–except kan implementeres som try–catch.

TryExcept har medlemmet body, som er setningene i try-blokken. Slenger disse en exception, letes det etter en passende håndterer i handlers. Handler er en liste av tupler, og disse ser slik ut:

```
Handler = [0]=Exp /*type*/, [1]=AssLvalue(flags=OP_ASSIGN), [2]=Stmt
```

Handler inneholder en type, som må matche den slengte type og et navn som kan tilordnes denne typen. Det tredje elementet er kodeblokken.

Hvis else_ er satt, vil dette kunne kodes i C++ ved at else_ etterfølger siste catch. Deretter genereres en unik label, og alle catch-blokkene avsluttes med en goto til denne.

TryFinally

```
TryFinally := body=Stmt, final=Stmt
```

Finally er en ting som ikke eksisterer i C++. Ting tyder på at neste utgave av standarden vil inkludere finally. Dagens C++ inkluderer ikke finally fordi C++ garanterer at destructors blir kalt på alle objekter når de går ut av skop, og poenget med finally er å rydde opp.

Jeg satser på å implementere finally ved å ta med koden for finally-blokken to ganger: først i en catch-all-blokk (`catch(...)`) med en throw uten argumenter til slutt i blokken. Deretter gjentas koden etter catch-blokken.

Python kunne inntil versjon 2.5 ikke ha både except og finally. I versjon 2.5 implementeres disse to eksemplene identisk.

```
try:
  print "foo"
except:
  print "bar"
else:
  print "baz"
finally:
  print "fubar"
```

```
try:
  try:
    print "foo"
  except:
    print "bar"
  else:
    print "baz"
finally:
  print "fubar"
```

While

`While := test=Exp, body=Stmt, else\_=(Stmt|None)`

While er seg selv lik i C++ og i Python når else-blokken ikke er til stede, og da oversetter jeg den på enklest mulig måte.

Else-delen kan behandles på grunnlag av at disse eksemplene er semantisk identiske.

```
while test:
  print "foo"
else:
  print "bar"
```

```
if test:
  while test:
    print "foo"
else:
  print "bar"
```

I tillegg kan man bruke do–while i C++-versjonen for ikke å teste mer enn nødvendig.

With

`With := expr, vars, body`

With er ny i Python 2.5, men kommer ikke til å støttes av kompilatoren med det første.

Yield

```
Yield := value
```

C++ har ingen støtte for yield, og derfor vil jeg ikke implementere dette.

A.1.2 AssLvalue

```
AssLvalue =
  AssAttr |
  AssList |
  AssName |
  AssTuple |
  Slice |
  Subscript
```

AssLvalue befinner seg gjerne på venstre side av tilordningsoperatoren eller som argument til del. Denne forskjellen synes i medlemmet flags som da vil være satt til henholdsvis OP_ASSIGN og OP_DELETE.

AssAttr

```
AssAttr := expr=Exp, attrname=Id, flags=(OP_ASSIGN|OP_DELETE)
```

AssAttr brukes når medlemsvalgoperatoren brukes i lvalue (med unntak av AugAssign) eller i del. Medlemmene i AssAttr-objektet er ellers expr og attrname, hhv. uttrykket til venstre for punktumet og navnet til høyre.

Etter Python-regler er det mange forskjellige muligheter her. AssAttr kan medføre at diverse medlemsfunksjoner hos en eventuell klasse på venstre side kalles. Hvordan AssAttr skal oversettes, avhenger derfor av typen til venstre side av punktum! Ikke alle av disse kan oversettes.

AssList

```
AssList := nodes=AssLvalue+
```

AssList forekommer kun når listekonstruksjoner forekommer på venstre side. Jevnfør dette med AssTuple.

AssName

```
AssName := name=Id, flags=(OP_ASSIGN|OP_DELETE)
```

Alle navn på venstre side av tilordningsoperatoren, dukker opp som AssName. I Python vil en AssName, hvis variabelen ikke er markert global, opprette et nytt symbol, og eksekveringen av AssName, vil tilordne et nytt symbolet et nytt objekt. I Python vil dette tilordne den underliggende referansen, og AssName brukes derfor ikke i forbindelse med andre tilordningsoperatorer, som +=, som kaller en medlemsfunksjon. Dette er et punkt hvor Python og C++ er forskjellig.

AssTuple

`AssTuple := nodes=AssLvalue+`

Etter syntaksanalysen, er AssList og AssTuple identiske. Semantisk vil begge utføre det samme: elementvis tilordning. I Python kan man ha et stort tre på venstre side, og man må da ha det samme på høyre side. Dette virker ved å legge virkningen rekursivt.

Subscript

`Subscript := expr=Exp, subs=Index+, flags=(OP_ASSIGN|OP_DELETE|OP_APPLY)`

Subscript er en fellesklasse for uttrykk som i grammatikken heter «subscription» og «longslice», dvs. at man har uttrykk på formen «expr[subs]». Dermed har jeg også presentert hva expr og subs representerer: henholdsvis objektet som det opereres på, og en liste over indekser/slices. Subscript brukes på flere måter, og flags bestemmer hvordan. OP_ASSIGN og OP_DELETE gjør det samme som før, mens OP_APPLY brukes når Subscript er del i et uttrykk (inkludert venstresiden i AugAssign).

I Python vil «Subscript» medføre at en av __getitem__, __delitem__ eller __setitem__ kalles med expr som self og subs.

Listen subs kan bestå av følgende syntaktiske elementer:

```
Index =
  Ellipsis |
  Exp |
  Sliceobj
```

Exp er et vanlig Python-uttrykk.

Ellipsis `Ellipsis :=`

Ellipsis har ingen medlemmer. Den markerer uttrykket «. . . ». I Python ville den sendt med __builtins__.Ellipsis som argument til __getitem__, __delitem__ eller __setitem__.

Sliceobj `Sliceobj := nodes=Longslice`

`Longslice = [0]=Exp(start), [1]=Exp(stop), [2]=Exp(stride)`

Sliceobj blir her laget av uttrykk av typen «start:stop:step». Hvis man utelater en av disse, settes None inn. I Python vil Sliceobj mappes til et slice-objekt.

Slice

`Slice := expr=Exp, lower=Exp, upper=Exp, flags=(OP_ASSIGN|OP_DELETE|OP_APPLY)`

Slice er et spesialtilfelle av Subscript når uttrykket ser slik ut: «expr[lower:upper]». En Slice(expr, flags, lower, upper) er ekvivalent med Subscript(expr, flags, [Sliceobj(lower, upper, None)]) i semantisk betydning.

A.1.3 Uttrykk

```
Exp =
  Add |
  Sub |
  Mul |
  Div |
  FloorDiv |
  Mod |
  And |
  Or |
  Backquote |
  Bitand |
  Bitor |
  Bitxor |
  CallFunc |
  Compare |
  Const |
  Dict |
  GenExpr |
  ListComp |
  Getattr |
  IfExp |
  Invert |
  Lambda |
  LeftShift |
  RightShift |
  List |
  Name |
  Not |
  Power |
  Slice |
  Subscript |
  Tuple |
  UnaryAdd |
  UnarySub |
```

Uttrykk brukes i setninger, og er i så måte underordnet – også fordi setninger ikke brukes i uttrykk. Som i C++ kan et uttrykk utgjøre en hel setning, men i denne AST-grammatikken vil setningsobjektet Discard da brukes. Dermed har AST-grammatikken et klart skille mellom uttrykk og setninger.

Add

```
Add := left=Exp, right=Exp
```

Add representerer operatoren +. I Python vil «x+y» konverteres til: Hvis x.__add__ finnes, kall denne med argument y. Hvis y.__radd__ finnes, kall denne med argument x. Ellers slenges en TypeError.

I C++ må kompilatoren på forhånd vite om uttrykket virker. Som syntaktisk sukker, kan også C++-typer overlaste operatoren +, men denne er innebygget for de innebyggede typer. Pga. hastighet, er det antageligvis mest fornuftig å satse på å oversette + med +, og tilpasse biblioteket til kravene.

Sub

```
Sub := left=Exp, right=Exp
```

Sub representerer operatoren −, altså subtraksjon. Ellers er den nærmest identisk med Add.

Mul

```
Mul := left=Exp, right=Exp
```

Mul representerer multiplikasjon. Ellers er den nærmest identisk med Add.

Div

```
Div := left=Exp, right=Exp
```

Div representerer en divisjon. Ser man på divisjon i forhold til de andre regneartene, kan divisjon slenge en DivisionByZeroError. I C++ divisjon med null udefinert, og hva som helst kan skje. Enkelte plattformer definerer det, dog, så da kan man ta hensyn til dette i kompilatoren. I utgangspunktet tar jeg ikke notis av dette.

En annen ting med Div er at den kan gjøre resultatet til et flyttall, selv med heltallsargumenter. Dette er varslet i fremtidig Python, og kan slås på ved å hente inn «division» fra __future__.

Mod

```
Mod := left=Exp, right=Exp
```

Mod representerer modulusoperasjonen. Modulus krever heltallsargumenter i Python som i C++.

And

```
And := nodes=Exp+
```

And representerer en logisk og. Det eneste spesielle med denne AST, er at argumentene sendes med som en liste.

Forskjellen mellom Python og C++ her, er at returverdien er verdien av det første argumentet som evalueres til usant eller det siste argumentet. I C++ vil operasjonen alltid returnere en bool.

Or

`Or := nodes=Exp+`

Or er en logisk eller. Den minner om logisk eller i C++, men uttrykkets verdi er i Python det første av uttrykkene som evalueres til sann. Ellers minner Or veldig om And.

Backquote

`Backquote := expr=Exp`

Backquote er en lite brukt konstruksjon, og det er en god grunn til det: Den er ekvivalent med et repr-kall, med expr som argument.

Bitand

`Bitand := nodes=Exp+`

Bitand representerer bitvis og. Operasjonen begynner med de to første argumenter i nodes, og reduserer argumentlisten til én verdi ved å bruke bitvis og på den gjeldende verdi og neste verdi. Operasjonen bitvis og kaller i Python __add__-metoden i første operand med den annen som argument eller __radd__-metoden til den annen operand med den første som argument.

Bitor

`Bitor := nodes=Exp+`

Bitor representerer bitvis eller. Som med Bitand, reduseres listen med argumenter ved å applisere operasjonen på de to første elementer, og senere på ett og ett av de resterende. I Python gjøres dette ved at en av metodene __or__ eller __ror__ kalles.

Bitxor

`Bitxor := nodes=Exp+`

Bitxor står for bitvis ekslusiv eller. Denne virker akkurat som Bitand og Bitor, men metoden som kalles er __xor__ eller __rxor__.

CallFunc

`CallFunc := node=Exp, args=Exp*, star_args=(Exp|None), dstar_args=(Exp|None)`

CallFunc gir et funksjonskall. Uttrykket node evalueres til objektet som kalles. Som regel vil node være et Name-objekt. Jeg vil først og fremst konsentrere meg om å få kall av navngitte funksjoner til å fungere.

Medlemmene args, star_args og dstar_args inneholder argumentene. Args gir her en tuppel med argumenter til funksjonen. Star_args og dstar_args har ingen ekvivalenter i C++, og det ville være mye jobb å implementere disse.

Compare

```
Compare := expr=Exp, ops=CmpOp+
CmpOp = [0]=Operator [1]=Exp
Operator = operator string
```

Compare er i denne AST-modellen en merkelig sak. Compare er en samleklasse for alle operasjoner som dekkes av __cmp__-metoden i Python, dvs $<$, $<=$, $>$, $>=$, != (og $<>$) og $==$, og av operatorene «in», «not in», «is» og «is not».

Strukturen i Compare-objektet er at expr representerer venstresiden i operasjonen lengst mot venstre, ops[0][1] gir dennes høyreside og ops[0][0] er en tekststreng som inneholder operatoren. Compare kan inneholde et vilkårlig antall sammenligninger, og hvert element i listen er et par av operatorer, ops$[n]$[0], og høyresider, ops$[n]$[1], og ops$[n-1]$[1] brukes som venstreside. Hele Compare-uttrykket er en logisk og av alle underuttrykkene.

Compare har som regel bool-verdi.

Const

```
Const := value=Literal
```

Const representerer et uttrykk, hvis type og verdi er den samme som medlemmet value. Const forekommer når man skriver inn verdier i kildekoden, som f.eks. strenger og tall.

Dict

```
Dict := items=([0]=Exp, [1]=Exp)*
```

En Dict står for en dictionary literal. Dict inneholder medlemmet items, som er en liste med nøkkel–verdi-par som skal inn i dictionaryen. En Dict vil opprette et dict-objekt, som da vil være uttrykkets verdi.

FloorDiv

```
FloorDiv := left=Exp, right=Exp
```

Floordiv gir divisjon, men runder svaret ned til nærmeste heltall. Med heltallsargumenter tilsvarer dette vanlig heltallsdivisjon i C++. Med flyttallsargumenter tilsvarer FloorDiv divisjon hvor svaret rundes ned, men fortsatt flyttallssvar.

GenExpr

```
GenExpr := code=GenExprInner
GenExprInner := expr=Exp, quals=GenExprFor+
GenExprFor := assign=AssLvalue, iter=Exp, ifs=GenExprIf*
GenExprIf := test=Exp
```

GenExpr står for et generatoruttrykk. Et slikt uttrykk kan alltid erstattes med en mengde setninger, og Python gjør nettopp dette. I følgende setninger vil funksjonen og uttrykket utenfor gi samme verdier.

(10∗x + y **for** x **in** range(10) **for** y **in** range(10) **if** y % 2)

```
def tmp():
    for x in range(10):
        for y in range(10):
            if y % 2:
                yield 10∗x + y
```

En GenExpr returnerer et generatorobjekt som kalkulerer neste verdi ved behov.

ListComp

```
ListComp := expr=Exp, quals=ListCompFor+ ListCompFor := assign=AssLvalue, iter=Exp, i
ListCompIf := test=Exp
```

En ListComp virker akkurat som den lignende GenExpr, bortsett fra at verdien av uttrykket er en list med verdiene fremfor et generatorobjekt.

Getattr

```
Getattr := expr=Exp, attrname=Id
```

En Getattr returnerer verdien av medlem ved navn attrname i objektet gitt ved uttrykket expr.

IfExp

```
IfExp := test=Exp, then=Exp, else_=Exp
```

If-uttrykket er nytt i Python 2.5, men det er enkelt å oversette: Det tilsvarer «?:» i C++.

Invert

```
Invert := expr=Exp
```

Invert returnerer den inverse, som er gitt ved operatoren ∼ i både Python og C++.

Lambda

```
Lambda := argnames, defaults, flags, code=Exp
```

Lambda-uttrykk returnerer enkelt og greit et funksjonsuttrykk. I C++ må man lage egne funksjoner for slike.

LeftShift

```
LeftShift := left=Exp, right=Exp
```

LeftShift representerer operatoren «<<» både i C++ og i Python. I C++ brukes ofte denne operatoren til formatert utskrift til fil.

RightShift

`RightShift := left=Exp, right=Exp`

RightShift representerer operatoren «>>» som virker på omtrent samme måte i Python og C++.

List

`List := nodes=Exp*`

List representerer en list display. Det finnes i utgangspunktet ingen ekvivalent i C++.

Name

`Name := name=Id`

Name returnerer objektet det refererer til.

Not

`Not := expr=Exp`

Not er en boolesk negasjon. Verdien er i Python en bool, akkurat som i C++.

Power

`Power := left, right`

Power finnes ikke i C++, men den finnes fra «<cmath>» med passelige overlastinger. Den eneste mangelen som vanskelig kan oversettes med dette, er at C++ ikke har noen heltallsversjon av std::pow().

Tuple

`Tuple := nodes=Exp*`

Tuple er et uttrykk som gir en tuppel med verdien av uttrykkene i nodes som medlemmer.

UnaryAdd

`UnaryAdd := expr=Exp`

Dette er unær pluss. I all praksis er denne en null-operasjon, men tatt med for å være symmetrisk med unær minus. Den kan i tillegg overlastes både i C++ og i Python.

UnarySub

`UnarySub := expr=Exp`

Unær minus, eller negasjon.

B Kildekode

Systemets kildekode er levert i elektronisk form med oppgaven. En versjon er også lagt ut på *http://hjem.ifi.uio.no/ roberth/pyemc/*.