GÖTEBORGS UNIVERSITET CHALMERS

Ett Python 3-frontend till Guile

Kandidatarbete inom Data- och informationsteknik

Stefan Kangas <skangas@skangas.se>

Per Reimers <per.reimers@gmail.com>

David Spångberg <davspa@student.gu.se>

Krister Svanlund <svanlund@student.chalmers.se>

Institutionen för Data- och informationsteknik

CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET

Göteborg, Sverige 2012
Kandidatarbete/rapport nr 2012:25

Abstract

GNU Guile is a virtual machine implementing Scheme. One of its goals is to be easy to use by other applications as an extension language. Since some time support for other languages than Scheme has been in development, for example ECMAScript and Emacs LISP. This report describes how one can implement Guile support for Python 3. A high level of integration with the Scheme support is emphasized. Since GNU Guile is Free Software some aspects of this that affects how a useful work has to be done are discussed. Both general problems, and problems that are associated with Guile, which arise when implementing Python 3 are discussed. Lastly, the possibility is explored, to use a Meta Object Protocol (MOP) to represent Python objects in GOOPS, Guiles main module for object oriented programming.

Sammanfattning

GNU Guile är en virtuell maskin som implementerar Scheme. Det har som mål att vara enkelt att användas av andra program som extensionspråk. Sedan en tid har det utvecklas stöd för andra språk än Scheme, exempelvis ECMAScript och Emacs LISP. Föreliggande rapport beskriver hur man kan implementera ett Guile-stöd för Python 3. Stor vikt läggs vid en hög nivå av integrering med Scheme-stödet. Eftersom GNU Guile är fri programvara diskuteras även vissa aspekter av detta som påverkar hur ett användbart arbete måste genomföras. Såväl generella problem, och sådana som är förknippade med Guile, som uppstår vid implementering av Python 3 behandlas. Slutligen utforskas även möjligheten att använda metaobjektprotokoll, MOP, för att representera Python-objekt i GOOPS, Guiles huvudsakliga modul för objektorienterad programmering.

Innehåll

1 Inledning

Stiftelsen Free Software Foundation (FSF) bildades i mitten av 1980-talet för att försvara och sprida fri programvara. Med fri programvara avser FSF programvara som ger användaren fyra friheter:

- Rätt att köra programmet i vilket syfte som helst. (Frihet 0)
- Rätt att studera hur programmet fungerar och modifiera det som man vill. (Frihet 1) Detta kräver tillgång till källkoden.
- Rätt att distribuera kopior av programmet. (Frihet 2)
- Rätt att distribuera kopior av en modifierad version av programmet. (Frihet 3)

En av Free Software Foundations huvudsakliga uppgifter är att dirigera utvecklingen av ett Unix-liknande operativsystem som uteslutande består av fri programvara. Detta operativsystem fick namnet GNU, och är mest känt i sin kombination med Linux-kärnan: det heter då GNU/Linux. [Sta02]

GNU Guile är namnet på det projekt som utvecklar Guile. Det är ett officiellt GNU-projekt, vilket betyder att det anses ingå i det fria operativsystem som FSF utvecklar. Guile är en virtuell maskin (VM) som implementerar Lisp-dialekten Scheme, ett minimalistiskt programmeringsspråk, som även är det officiella extensionspråket för operativsystemet GNU. Att använda Guile som extensionspråk betyder att man inkluderar Guile i en annan applikation, och därigenom får möjlighet att modifiera applikationens beteende genom att skriva kod som Guile kan tolka och exekvera. Den som skriver en applikation som inkluderar Guile ansvarar för att tillhandahålla ett lämpligt API. Guile är specifikt utformat för att vara enkelt att bädda in som extensionspråk i andra program men kan även användas fristående.[1]

Scheme tillhör familjen Lisp-1-språk vilket innebär att det har ett gemensamt namespace[2] för funktionsnamn och variabelnamn. En av de mest framstående koncepten hos Lispspråken är deras kraftfulla makrosystem, och speciellt för just Scheme är dess hygieniska makron.[3] Guile implementerar den version av Scheme som beskrivs av Revised[5] Report on the Algorithmic Language (förkortas R5RS) [ABB+98] samt innehåller egna tillägg, till exempel ett modulsystem, stöd för POSIX, trådar och nätverksprogrammering.

[1] Se vidare projektets webbplats https://www.gnu.org/software/guile/.

[2] Se ordlista i Appendix A.

[3] Ett makro beskriver hur en transformation av Lisp-kod ska göras och utförs innan koden kompileras. Detta möjliggör för programmeraren att skapa egna språkkonstruktioner och är en viktig metod för att undvika kodduplicering i Lisp-språk. Hygieniska makron innebär att de, till skillnad från till exempel makron i språket C, inte påverkas av, kan skriva över eller överskugga variabler som finns i den miljö makrot blir expanderat i.

1.1 Bakgrund

> "Compilers are for hacking, not for admiring or for complaining about. Get to it!"
> – Guile Reference Manual, section 10.4.8

Projektet GNU Guile startades 1992 och har redan från början haft som uttalat mål att stödja flera olika språk. Guile består sedan en tid av flera olika kompilatorer som är strukturerade i ett kompilatortorn. I och med implementeringen av detta kompilatortorn fick Andy Wingo, en av projektets huvudutvecklare, en insikt. För att lägga till fler språk till Guile behöver man enbart kompilera till ett valfritt språk i detta torn, så kommer översättningen nedåt i hierarkin fungera automatiskt. Detta gjorde att det blev möjligt att bygga helt nya frontends.

> "I've been hacking on a compiler for Guile for the last year or so, and realized at some point that the compiler tower I had implemented gave me multi-language support for free.
>
> But obviously I couldn't claim that Guile supported multiple languages without actually implementing another language, so that's what I did." [Winb]
>
> – Andy Wingo

Ett frontend brukar man kalla den del av kompilatorn som översätter från ett programmeringsspråk till en mellanliggande representation. Detta kan skiljas från en backend, som översätter från en mellanliggande representation till körbar kod. Körbar kod ska här förstås brett; det kan till exempel vara fråga om kod som ska köras på en VM. Detta är precis fallet med GNU Guile. Fördelen med att kompilera till en mellanliggande representation i stället för direkt till körbar kod är att man inte behöver bekymra sig om alla detaljer på en gång. Man kan hålla sig på en högre abstraktionsnivå, och på så sätt kan flera frontends dra fördel av samma backend. Man kan även ha flera backends.

Sedan 2009 har Guile ett rudimentärt stöd för ECMAScript 3.1. Det finns nu flera språk som befinner sig i olika utvecklings- och planeringsstadier, såsom Lua och PHP.[4] Ett av de mer högprioriterade språken under utveckling, som varit föremål för två Google Summer of Code-projekt, är Emacs Lisp, vilket är den största delen av den populära texteditorn GNU Emacs. Förhoppningen är att på sikt kunna byta ut den vanliga Emacs Lisp-motorn i Emacs till Guile. Om detta kommer bli verklighet återstår att se.

De senaste åren har Guile under en ny ledning tagit kvalitativa steg framåt, och stora planer är på gång. Exempelvis har stödet för andra språk utvecklats, flera standardiserade moduler från R5RS implementerats och stora förändringar har gjorts för att öka den virtuella maskinens prestanda.

Vad betyder det att man implementerar ett nytt frontend? Fallet med Emacs Lisp är lite speciellt eftersom det är så tätt knutet till ett specifikt program, men när det kommer till enbart själva språkstödet så fyller det viktiga funktioner. Ett välimplementerat stöd innebär att det omedelbart blir mer attraktivt för programutvecklare att använda Guile som extensionspråk. Det utökar omedelbart den grupp som skulle kunna tänka sig att skriva tillägg, till den grupp av användare som

[4] http://savannah.nongnu.org/projects/guile-php

kan programmera det nya språket. Projekt som använder Guile kan på detta sätt dra nytta av uppfinningsrikedomen bland många fler användare. Därigenom blir Guile mer användbart.

Python är ett populärt programmeringsspråk med flera implementationer. Python brukar ibland framhållas som ett exempel på ett särskilt väldesignat språk. Vad man än tycker om den saken är det i vart fall möjligt att implementera, vilket inte minst visar sig av att det redan existerar flera framgångsrika implementationer. Språket har en tillgänglig dokumentation.

Den senaste versionen av Python är version 3.[5] Versionen är inte helt bakåtkompatibel med version 2. Stödet och användandet för Python 3 ökar långsamt; trots att det är över tre år sedan version 3 släpptes är fortfarande Python 2-kod det vanligaste. En orsak är antagligen att Pythons popularitet ökade kraftigt under version 2 och att många populära bibliotek som inte aktivt underhålls utvecklades under den tiden. För att driva utvecklingen av Python framåt och på grund av den utveckling som skett mellan versionerna 2 och 3 rekommenderas officiellt användandet av Python 3 så långt det är möjligt. [Wik]

Alla dessa saker sammantaget gör det attraktivt och intressant att arbeta på en Python 3-implementation för Guile.

1.2 Syfte

Syftet med kandidatarbetet är att implementera ett Python 3-frontend till GNU Guile. Detta frontend ska hantera alla delar från användarens inmatning till, av Guile, körbar kod: lexning, parsning och kodgenerering.

Implementationen ska vidare hålla tillräckligt god kvalitet för att i princip kunna inkluderas i den officiella versionen av GNU Guile. Detta innebär självklara saker som att koden ska vara väldokumenterad och strukturerad. En test suite ska säkerställa en viss kvalitet på koden och förhindra regressioner.

1.3 Problem

På en övergripande nivå kan man säga att en kompilator består av tre olika steg. Det första steget är lexning, där man bryter ned en teckensträng i beståndsdelar som identifierare, tal och textsträngar. Därefter följer parsning, vilket betyder att utifrån en given grammatik bygga upp ett abstrakt syntaxträd (AST).[6] Slutligen översätts detta syntaxträd till ett målspråk. Dessa tre utgör varsitt problemområde, även om lexning och parsning i allmänhet brukar behandlas som en enhet.

Översättningen från Python till ett målspråk kan brytas ned i en rad mindre problem: att hantera objekt, skillnader i typer mellan Python och Guile, generatorer, funktioner och argument (inklusive default-värden och keyword-argument), inbyggda funktioner, undantag (exceptions), vissa

[5] Vi skriver Python 3 eller bara Python i stället för den vanliga beteckningen Python 3.x. Med detta avses den senaste versionen av Python 3.x, i skrivande stund Python 3.2.3 som släpptes den 14 april 2012.

[6] Det syntaxträd som används för detta arbete finns dokumenterat i appendix C.

datatyper (listor, tuplar, mängder, icke-modifierbara och modifierbara), list comprehensions. Implementationen av Pythons standardbibliotek är ett separat problem.

Vidare ingår löpande testning som ett delproblem. Här kan man ha nytta av Pythons egna test suite, Guiles test suite samt egenutvecklade tester.

1.3.1 Integrering

I Guiles Read-Eval-Print Loop (REPL) kan man byta språk interaktivt genom kommandot `,language <språk>`. Vi har strävat efter att integrera Python 3-stödet så att kommandot `,language python3` ger en REPL där man interaktivt kan köra Python 3-kod. Syftet är samma nivå av integration som det redan befintliga stödet för ECMAScript.

Ett exempel på hur en testkörning med ett fullt integrerat Python 3 skulle kunna se ut:

```
scheme@(guile-user)> ,language python3
Guile Python 3 interpreter 0.1-dev on Guile 1.9.0
Copyright (C) 2001-2012 Free Software Foundation, Inc.
Enter ',help' for help.
python3@(guile-user)> "Hello world! " + str(99)
$1 = 'Hello world! 99'
python3@(guile-user)> [1,2,3][:-1]
$2 = [1, 2]
python3@(guile-user)> [n**2 for n in range(10) if n % 2 == 0]
$3 = [0, 4, 16, 36, 64]
```

1.3.2 Målspråk

För att förstå hur Guiles kompilatortorn fungerar i mer detalj kan man exemplifiera med Scheme, som är Guiles standardspråk. Scheme kompileras i första steget till Tree-IL. Tree-IL har stora likheter med Scheme, och är i själva verket macro-expanderad Scheme, alltså precis det som fås av att köra Guiles funktion `macroexpand` på Scheme-kod. `macroexpand` expanderar macron och hanterar till exempel block och lexical scoping genom att döpa om variabler till unika namn. Tree-IL kompileras sedan till GLIL, som är ett strukturerat språk vars uttryck har större likheter med Guiles VM än med Scheme. Detta kompileras till Guile VM-assembler, som i ett sista steg kompileras till bytekod och objcode.[7] Alternativt kan det hela sedan kompileras ner till ett målspråk, Value, som egentligen inte är ett språk alls utan bara ett "dummy"-mål för att exekvera koden och returnera värdet.

För att implementera en ny frontend till Guile finns det flera möjliga nivåer i detta kompilatortorn som implementationen kan sikta på. Mot bakgrund av bland annat en rekommendation av en av Guiles huvudutvecklare Andy Wingo, valdes Tree-IL som målspråk.

[7] objcode är en Guile-specifik binär representation för bytekod.

Direkt ovanför Tree-IL i hierarkin ligger endast källspråk som Scheme eller ECMAScript. Eftersom dessa språk kan ha egenskaper som riskerar att komma i vägen eller förbli helt oanvända finns det ingen anledning att välja något av dessa som målspråk. Tvärtom verkar det snarare kunna leda till begränsningar och svårigheter.

Direkt nedanför Tree-IL i Guiles språkhierarki ligger GLIL. Det verkar olämpligt att välja detta eftersom det ligger på alltför låg nivå. Flera av de resurser man kan använda om man väljer Tree-IL som målspråk saknas för GLIL. Ett exempel är dokumentation i form av implementationer av andra språk.

1.3.3 Semantiska skillnader

Även om Scheme och Python har flera likheter finns det stora och ofrånkomliga skillnader. Dessa kräver viss eftertanke för att kunna hanteras utan att prestandan blir alltför lidande. Målet bör hela tiden vara att eftersträva en hög grad av likhet med andra implementationer av Python 3.

Flyttal och liknande kan ge upphov till skillnader i avrundning men så länge en rimligt hög precision kan upprätthållas finns det ingen större anledning för detta projekt att lägga tid på att få dessa helt ekvivalenta. Datatyper, objektorientering och generatorer är några andra exempel på områden där det finns avgörande skillnader, vilket kommer att visas i senare kapitel.

Scheme har precis som de flesta andra Lisp-språk ett numeriskt torn för att representera olika sorters tal. Detta innebär att tal representeras av en tornliknande struktur där varje nivå är av en högre precision; `1` är en `integer`, som är en `rational`, som är ett `real number` och så vidare. Genom denna struktur kan tal alltid omvandlas neråt i strukturen utan att förlora precision; en `integer` kan översättas till en `rational` med nämnaren `1` men en `rational` (till exempel `3/2`) kan inte, generellt sett, översättas till en `integer` (`3/2` är inte ett heltal) utan att förlora precision.[8]

Python har också ett numeriskt torn, formellt implementerat i modulen `numbers`, som liknar den som finns i Guile. Dock förlitar sig Python på mer hårdvarunära representationer av tal när det är möjligt och vid tolkning av literala tal i källkod.

1.3.4 Sociala och juridiska aspekter på att få koden inkluderad

Att få koden accepterad, det vill säga inkluderad i den officiella versionen, torde vara målet för varje allvarligt menat försök att bygga ny funktionalitet till en fri programvara. Det finns såväl juridiska som sociala aspekter på detta. Låt oss börja med de juridiska.

Majoriteten all fri programvara är licensierad under en licens som heter GNU General Public License (GPL). Av alla fri programvaru-projekt är det 42 procent som använder GNU GPL medan den närmaste efterföljaren (MIT License) används av endast 12 procent. [Sof] GPL är en så kallad

[8] Dokumentation för Guiles numeriska torn kan hittas på http://www.gnu.org/software/guile/manual/guile.html#Numerical-Tower

copyleft-licens, vilket innebär att den inte bara garanterar användaren de fyra friheter som beskrevs i inledningen, utan också att det ställs krav på att den som sprider modifierade versioner själv måste släppa sina ändringar under samma villkor. [Stab] Utvecklaren kan ha flera skäl till att vilja detta: ett är givetvis att garantera att programvaran förblir fri. På ett större plan är det också ett sätt att bjuda in andra att skriva fri programvara: "om du bara släpper din programvara fri, får du även lov att använda denna kod". [Staa]

För att kunna uppfylla de krav som copyleft ställer måste en ändring på ett sådant program vara licensierad under en kompatibel licens. En utvecklare kan själv välja att släppa koden under en given licens, och därmed lösa problemet. Om däremot koden hämtats någon annanstans, exempelvis från ett annat program, måste man försäkra sig om att den koden verkligen är licensierad under en kompatibel licens.

Free Software Foundation har ett ännu strängare förfarande vad gäller bidrag till projekt som de själva direkt hanterar. De kräver att utvecklare överlåter sin upphovsrätt till stiftelsen, eftersom de menar att det ger starkare garantier för att programvaran förblir fri. Detta ska nämligen ge FSF bättre möjlighet att försvara licensavtalet mot avtalsbrott i domstol. Om endast en juridisk person kontrollerar upphovsrätten blir det enklare att processa i nordamerikanske rätt, då de annars måste inhämta godtycke från samtliga rättighetshavare.

> "[I]n order to be able to enforce the GPL most effectively, FSF requires that each author of code incorporated in FSF projects provide a copyright assignment, ... That way we can be sure that all the code in FSF projects is free code, whose freedom we can most effectively protect, and therefore on which other developers can completely rely." [Mog]

För att koden verkligen ska kunna inkluderas måste därför de som skriver koden överföra sin upphovsrätt till FSF. Detta sker genom ett blankettförfarande. En konsekvens av detta är att det begränsar vilken annan kod och programvara som går att inkludera i ett bidrag till GNU Guile. Även om programvaran är fri kommer projektet endast inkludera den om FSF förfogar över dess upphovsrätt.

Vid sidan av juridiska finns det också sociala aspekter kring att få koden accepterad i ett fri programvaru-projekt. Det gäller att övertyga utvecklare om att det är tekniskt lämpligt att inkludera koden. Här räcker det inte med att vara rätt ute. Man måste också kunna visa detta på ett sätt som andra utvecklare kan förstå och hålla med om. Man måste alltså övertyga andra utvecklare om detta, i synnerhet den eller de ledande utvecklarna. En buggfix måste till exempel demonstreras vara felfri och tekniskt välgrundad, och när det som här är fråga om en ny funktion måste den demonstreras vara önskvärd och i själva utförandet tillräckligt välgjord.

Att introducera nya beroenden av bibliotek, programvaror och dylikt är även förknippat med en teknisk underhållsbörda. Det skulle direkt påverka alla distributörer och användare av Guile, och kräva av dem att de håller sagda externa beroenden uppdaterade med de senaste säkerhetsuppdateringarna. Att undvika detta ökar troligtvis sannolikheten att koden bedöms lämplig för inkludering.

Av dessa skäl har vi som genomfört arbetet under dess gång upprätthållit kontakt med Guile-

utvecklarna. Detta har främst skett genom IRC[9], men saker relaterat till vårt arbete har även diskuterats på projektets e-postlistor. Detta har varit ett bra sätt att få värdefull input och idéer på delar av projektet, och hör väl ihop med en kollaborativ utvecklingsprocess inom fri programvara. Detta ger också arbetet en förankring inom projektet. Tips och hjälp utifrån har, i den mån den varit av större betydelse, dokumenterats i denna rapport.

1.4 Avgränsningar

Den viktigaste och största delen av arbetet är implementationen av semantiken i Python 3. Med semantik avses innebörden av en giltig språkkonstruktion. Detta är att skilja från syntax, som anger vad som över huvud taget är en giltig språkkonstruktion.

Att implementera hela Python 3 låter sig inte göras inom ramen av ett kandidatarbete, inte minst på grund av tidsbegränsningen. I stället är meningen att producera en stabil grund för fortsatt utveckling. Denna grund kan omfatta en rimligt stor delmängd av Python.[10]

Även om delar av standardbiblioteken är nödvändiga för att få igång grundläggande funktionalitet är meningen inte att implementera hela Pythons standardbibliotek. Speciellt bör de delar helt undvikas som uppenbart har karaktären av valfria tillägg. Dessa tar alltför mycket tid och resurser och är på inget sätt kritiska för att implementationen ska vara användbar. Delar av standardbiblioteken borde dessutom med lätthet kunna inkluderas om det bara finns ett tillräckligt välfungerande språkstöd.

Man kan anta att det inte är realistiskt att helt följa Python-standarden på grund av skillnader i hur till exempel CPython och Guile fungerar när projektet skrivs på en så hög nivå som Tree-IL. Däremot måste avsteg dokumenteras och förklaras, särskilt i de fall där de inte lätt går att åtgärda senare.

Arbetet syftar inte till att bygga en test suite för Python 3 eller Guile, utan test suiten är enbart en hjälp under utvecklingen.

Även om prestanda är viktigt i alla verkliga applikationer följer arbetet i detta avseende Donald E. Knuths rekommendation:

> "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." [Knu74]

1.5 Översikt

I avsnitt 2 beskrivs språket Python. Några exempel ges som introduktion till språket, och vissa särskilt intressanta eller förvånande detaljer belyses. Detta avsnitt kan förmodligen hoppas över av den som känner till Python väl.

[9] IRC står för Internet Relay Chat. Se http://en.wikipedia.org/wiki/IRC för mer information.
[10] Gällande formuleringen "rimligt stor", se vidare avsnitt 5.

I avsnitt 3 beskrivs hur Python har implementerats i Guile. Detta avsnitt förutsätter vissa förkunskaper i Lisp-liknande språk, exempelvis Scheme.

I avsnitt 4 beskrivs till en början metaobjektprotokollet. Därefter diskuteras hur man hade kunnat använda det för att implementera Python. Detta avsnitt torde inte kräva några speciella förkunskaper.

Därefter följer en diskussion i avsnitt 5 och våra slutsatser i avsnitt 6.

Koden finns tillgänglig på https://github.com/skangas/guile/tree/wip-python3. De intressanta bitarna finns i `module/language/python3/`.

2 Språket Python

Detta avsnitt är en sammanfattning av vår undersökning av språket Python 3. Avsnittet kan tjäna som hjälp för någon som vill implementera Python 3.

2.1 Några exempel

Det kan vara lämpligt att börja med några exempel för att få en känsla för Python 3.[11] Nedan följer en rekursiv definition av Fibonacci-talen:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Denna implementation är enkel men samtidigt mycket ineffektiv. Då Python är ett imperativt språk är en iterativ lösning mer naturlig:

```
def fib(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

Den inbyggda funktionen `range(n)` skapar ett objekt som kan generera alla heltal mellan `0` och `n-1`. Denna konverteras till en iterator som `for`-loopen kan iterera över. Vi ser här att redan loopar beror mycket starkt på objekt.

Tilldelningen (`a, b = b, a + b`) fungerar så att det till vänster om kommat i vänsterledet tilldelas det till vänster om kommat i högerledet, och vice versa för det till höger om kommat. En intressant sak med denna konstruktion är att allting i högerledet evalueras först, vilket i detta fall besparar oss från att behöva använda en tillfällig variabel.

Här är en ineffektiv men enkel version av QuickSort som bygger på list comprehension[12]:

[11] Exemplen är hämtade från http://en.literateprograms.org/Fibonacci_numbers_(Python)
[12] https://secure.wikimedia.org/wikipedia/en/wiki/List_comprehension

```
def qsort(list):
    if not list:
        return []
    else:
        pivot   = list.pop()
        lesser  = qsort([x for x in list if x < pivot])
        greater = qsort([x for x in list if x >= pivot])
        return lesser + [pivot] + greater
```

List comprehensioner kan använda sig av filter[13] och nästlade loopvariabler (vilket ger den kartesiska produkten av de givna listorna, på samma sätt som i exempelvis Haskell).

Python 3 har en REPL där man kan programmera direkt; som exempel kan man använda denna för att definiera en klass:

```
>>> class A:
...     b = 1
...     def bar(self):
...         return self.b
...
```

Det användaren skriver markeras av `...` och >>>. Rader som saknar ett av dessa prefix är resultatet av en evaluering.

```
>>> A.b
1
>>> foo = A()
>>> foo.bar()
1
>>> foo.b = 2
>>> foo.bar()
2
>>> A.b
1
```

I detta exempel inspekterar vi bland annat klassattributet `b` och skapar ett objekt av typen `A`. Dessa exempel bör ge läsaren en känsla för språket Python.

2.2 Pythons specifikation

Pythons syntax och huvudsakliga semantik specificeras i *The Python Language Reference*. En påtaglig skillnad mellan denna dokumentation och exempelvis C++-standarden eller ECMAScript Language Specification, är att den inte är lika ingående eller detaljerad. Den är dessutom kortare, vilket beror på att Python i större utsträckning har valt att förlägga saker till sitt standardbibliotek.

[13] I exemplet ovan är till exempel `if x < pivot` ett filter.

The *Python Standard Library* beskriver standardbiblioteket, inklusive vissa valfria komponenter. Python bygger till stor del på just detta standardbibliotek. Vissa saker som man annars kanske kunde tänka sig skulle tillhöra själva språket dokumenteras här, så som inbyggda funktioner, konstanter och undantag. Också datatyper som tal, arrayer och strängar är förlagda till standardbiblioteket. Detta betyder att man inte kan undgå att implementera delar av standardbiblioteket om man vill få en användbar Python-implementation.

2.3 Pythons objekt

Allting i Python är objekt. Detta inkluderar moduler, funktioner och till och med klasser. Även andra språkstrukturer, såsom loopar och if-satser har en nära integrering med Pythons objektsystem.

Inom Pythonprogrammering används duck typing, vilket innebär att man betonar objekts beteende, snarare än den faktiska typen. Exempelvis kan en generator i många situationer användas som en lista då de efterliknar många av de beteenden som listor har.

2.4 Arv

Som objektorienterat språk har Python stöd för arv, vilket är relativt enkelt; en klass har en superklass, och om man försöker nå ett attribut som inte finns direkt i klassen så kollas superklassen i stället.

```
class A:
    def a(self):
        print("A.a(): anropar b()")
    def b(self):
        print("A.b()")

class B(A):
    def b(self):
        print("B.b()")

b = B()
```

I exemplet ovan kommer ett anrop `b.a()` anropa `A.a()` och `B.b()`, och inte `A.b()`.

Det som gör en implementation av Pythons objektsystem mer komplext är multipelt arv. Många av de problem som är rättframt att lösa i enkelt arv, får betydligt fler detaljerade nyanser i en flerarvsmiljö.

I koden nedan får klassen C ärva två konflikterande definitioner av metoden a:

```
class A:
    def a(self):
        print("A.a()")

class B:
    def a(self):
        print("B.a()")

class C(A, B):
    pass

c = C()
```

Om man anropar `c.a()` så anropas i första hand den första klassens metod, i detta fall `A.a()` i stället för `B.a()`.

2.5 Objektberoende språkkonstruktioner

Flera språkkonstruktioner för flödeskontroll beror på objektorientering. I Java är exempelvis typen på if-satsens testuttryck alltid den primitiva typen boolean. I Python däremot anropas testuttryckets `__bool__()`-metod[14] vilken i sin tur dynamiskt avgör objektets sanningsvärde. Alla kriterier för när ett värde är falskt beror alltså direkt på objektorientering. Även `False` och `True` är objekt – de tillhör klassen `Bool` och implementerar `__bool__`- metoden (vilket i detta fall är en identitetsfunktion).

Även for-loopar (vilket motsvarar for each-loopar i andra språk) använder ett objektorienterat sätt att iterera (det skapas en generator som man itererar med).

2.6 Parsning av Python

Det finns olika sätt att formulera en grammatik på, vilka har för- och nackdelar. Det vanligaste är Context-free Languages, vilket på svenska blir ungefär "kontextoberoende språk". Dessa uttrycks av en Context-free Grammar (CFG). Grammatiken brukar specificeras i allmänhet oftast på en särskild form: Backus-Naur Form (BNF) med Regular Expression-tillägg. Detta är också den konvention som följs i Python-dokumentationen.[15]

Python är ett indenteringsberoende språk. Detta betyder att block avslutas genom att minska indenteringen, och påbörjas genom att minska den. En konsekvens av detta är att man inte kan göra en kontextoberoende syntax för språket.

Det vanliga sättet att hantera detta verkar vara en preprocessor, där man måste hantera saker som logiska rader, kommentarer, indentering och avindentering (det vill säga block).

[14] Om `__bool__()`-metoden saknas används i stället `__len__()`-metoden som då returnerar ett int värde vars sanningsvärde används. Om även denna metod saknas ses objektet som True i if-satsen.

[15] http://docs.python.org/py3k/library/ast.html?highlight=ast#abstract-grammar

2.7 Namnrum och scope

Python har ett relativt okomplicerat system för scoping. Oftast brukar man tala om att det finns tre typer av scope i Python: ett scope för inbyggda metoder, ett scope för namn definierade i moduler och ett lokalt scope. [Foue]

En förvånande egenskap med Python är det ej är garanterat att ett block öppnar ett nytt scope. De variabler som skapas i ett block kan då exempelvis nås i en hel funktionskropp. Låt oss definiera en funktion `foo` i Pythons REPL för att demonstrera denna egenskap:

```
>>> def foo():
...     if True:
...         a = 5
...     return a
```

Här är motsvarande kod i programspråket C:

```
int main(void) {
  if (1) {
    int a = 5;
  }
  return a;
}
```

Om man försöker kompilera detta kodexempel får man följande fel:

```
$ gcc test.c
test.c: In function     main    :
test:5: error:     a     undeclared (first use in this function)
```

Felet beror på att `a` bara kan nås inuti det block där den definieras. I Python däremot kan man nå variabeln även utanför detta block, eftersom Python inte har lexical scoping utan metodscoping.

```
>>> foo()
5
```

Slutligen kan man lägga till saker från andra moduler till det nuvarande scopet genom att importera dem:

```
import lib.a
from lib.b import *
```

2.8 Diskussion

Det är inte alls självklart hur man delar upp vad som ska anses utgöra själva språket och vad som utgör ett standardbibliotek. Man kan till exempel hävda att ett språk utan ett standardbibliotek i sig är ganska meningslöst eftersom man inte kan göra speciellt mycket med det. I själva verket måste en uppdelning mellan språk och standardbibliotek innehålla ett visst mått av godtycklig bedömning.

Att utvecklarna lagt så mycket i standardbiblioteket hänger troligtvis samman med hur Python är uppbyggt med avseende på objektorientering.

2.9 Slutsatser

Python är ett mycket dynamiskt språk. Kostnaden är att det blir mycket komplext att implementera och den initiala tröskeln för att komma igång med en implementation får anses hög. Det första problemet man bör ta itu med är hur man hanterar objektorientering. Vidare är parsning ett stort potentiellt problem då Python i många avseenden är mer inriktat på att vara enkelt för användare, förlåtande och uttrycksfullt, egenskaper som gör språket komplicerat att parsa.

3 Python i Guile

Detta avsnitt beskriver hur vi konkret angripit problemet med att implementera Python 3 för Guile.

Det finns färdiga frontends till Guile för andra språk som man kan studera och lära sig av. Också Python finns i en rad olika implementationer, och eftersom de flesta av dem är fri programvara kan man studera deras källkod. Detta gör att även de med fördel kan användas som resurser för information och idéer.

Ett bra sätt att undvika regressioner och se när milstolpar har uppnåtts är en test suite. Som orakel[16] kan man med fördel använda CPython, eftersom det är den mest använda implementationen av Python idag. Inspiration för testfall kan hämtas från diverse andra resurser och projekt inom Python-världen. Guile har färdiga strukturer för unit testing i vilka testfallen bör integreras.

3.1 Lexning och parsning

Med hjälp av Pythons standardmodul `ast` har vi kunnat arbeta med Pythons abstrakta syntaxträd direkt, utan att behöva göra lexning och parsning själva. En parser implementerades som läser in källkod och konverterar denna till ett Python-objekt. Detta objekt omvandlas senare till en datastruktur som lämpar sig för Scheme. Nedan följer ett exempel på hur denna parser fungerar:

```
def main():
  print("Hello World!")
```

[16] Med orakel avses ett program som givet in- och utdata avgör om testet fallit väl ut.

Konverteras således till ett Python-objekt med strängrepresentationen:

```
Module(
  body=[
    FunctionDef(name='main',
                args=arguments(args=[],
                               vararg=None,
                               varargannotation=None,
                               kwonlyargs=[],
                               kwarg=None,
                               kwargannotation=None,
                               defaults=[],
                               kw_defaults=[]),
                body=[
                  Expr(value=Call(func=Name(id='print', ctx=Load()),
                                  args=[Str(s='Hello World!')],
                                  keywords=[],
                                  starargs=None,
                                  kwargs=None))],
                decorator_list=[],
                returns=None)])
```

Detta Python-objekt serialiseras sedan till en textsträng i ett format som Guile kan läsa in och avserialisera som en Scheme-lista. Denna lista kommer ha exakt den AST-struktur som definieras i appendix C. Ovanstående exempel avserialiseras då till följande struktur:

```
(<module>
  ((<function-def> main (() #f #f () #f #f () ())
                   ((<expr>
                     (<call>
                      (<name> print
                              <load>)
                      ((<str> "Hello World!"))
                      () #f #f))) () #f)))
```

3.1.1 Parsing Expression Grammars (PEG)

Pythons syntax är uttryckt som en Context-Free Grammar (CFG). CFG:s har vissa problem, varav ett är att de är tvetydiga: vissa sådana problem är så pass vanligt förekommande att de är kända som reduce/reduce-konflikter eller shift/reduce-konflikter. Det är i dessa fall fråga om att man under parsningen kan gå vidare på två olika sätt, och kan komma fram till fler än en syntaxträd för samma kod beroende på vilket val kompilatorn gör.

För att överkomma detta problem har en ny teknik utvecklats som heter Parsing Expression Grammar[17] (PEG). [For04] PEG är släkt med CFG, men de två teknikerna är inte ekvivalenta. För att

[17] https://en.wikipedia.org/wiki/Parsing_expression_grammar

undvika sådana konflikter som CFG kan råka ut för introducerar PEG ett sätt att prioritera mellan de olika reglerna, vilket gör att det endast finns ett giltigt syntaxträd, givet att indatan går att tolka enligt de givna reglerna.[For04]

Parsning av språket kan göras bland annat med ett befintligt bibliotek, en egen implementation av grammatiken, användande av parsergenerator, eller helt egenskriven kod. Det lämpligaste torde vara att använda någon av de inbyggda moduler för parsning och kompilering som redan finns i Guile.

Det har nyligen tillkommit en modul för att bygga parsers med PEG i Guile. Modulen är fortfarande under utveckling så för att testa har vi gjort försök att uttrycka delmängder av Pythons syntax i PEG. Detta har inte lett till några användbara resultat då flera buggar stöttes på vilka bedömdes ligga i själva PEG-modulen, och vars ursprung det inte fanns tid att spåra och åtgärda.

3.1.2 Preprocessor

För att göra det enklare att tolka Python gjorde vi en preprocessor. Denna ska främst hantera Pythons logiska rader. Den hanterar även det att markera när indentering av koden ökar eller minskar då Python använder det för att dela upp koden i block. På grund av detta beroende av indentering är det inte möjligt att parsa Python i "ren" CFG eller PEG då detta gör koden kontextberoende. Kontextberoende betyder alltså att innebörden av en kodrad tolkas olika beroende på innehållet i de föregående raderna.

Preprocessorn är uppbyggt som ett antal filter vilka sekventiellt filtrerar filen enligt vissa regler och sedan returnerar en ny ekvivalent version av koden.

```
(define (preprocessor str)
  (add-newline-and-indent-tokens
   (convert-tabs
    (remove-comments
     (handle-line-continuations
      (convert-triple-quotes
       str))))))
```

Denna kod ska läsas nedifrån och upp:

1. Konvertera strängar som sträcker sig över flera rader till vanlig strängsyntax.
2. Byt ut explicita radfortsättningar, som i Python markeras med \, på så sätt att dessa inklusive efterföljande nyradstecken tas bort.
3. Radera samtliga kommentarer.
4. Konvertera alla tabbar till mellanslag då de används som indentering.
5. Lägg till nyradstecken tokens och indent tokens. De förstnämnda markerar slutet på en logisk rad, medan de senare markerar öppnandet av nya block. Detta kompliceras något av att det kan finnas implicita radfortsättningar och att tomma rader måste ignoreras.

Preprocessorn missar i sin nuvarande implementation vissa edge-cases:

```
if True: pass
if True: pass ; True
```

Ovan är giltig Python-kod, men vår preprocessor ger inte korrekt output.

3.2 Kompilering av AST

När vi genererat den representation av Python 3 som diskuteras i föregående avsnitt är nästa steg kompileringen till Tree-IL. I filen `compile-tree-il.scm` återfinns den funktion som tar hand om detta:

```
(define* (comp x e #:optional toplevel)
  "Compiles the Python 3 expression X into a tree-il expression using
the environment E. The optional argument TOPLEVEL controls if
assignments should be bound in the toplevel or local scope."
  (pmatch x
    ((<module> ,stmts)
     (comp-block #t stmts e))
    [...]
    ((<return> ,exp)
     `(primcall return ,(comp exp e)))
    [...] ))
```

`pmatch` är ett Scheme-makro som inte helt oväntat mönstermatchar på sitt argument, i exemplet ovan `x`. `<module>` och `<return>` i koden ovan tolkas här som konstanta värden under matchning medans `,exp` och `,stmts` matchas mot godtyckligt giltigt Scheme uttryck. I fallet för `<return>` kommer först uttrycket `(comp exp e)` evalueras. Låt oss anta att det returnerar `(const 3)`. Då kommer det kompletta Tree-IL uttryck som returneras från `comp` vara: `(primcall return (const 3))`.

Nedan följer några fler funktioner som används flitigt i källkoden och även i några kodexempel:

```
(define-syntax-rule (-> (type arg ...))
  '(type ,arg ...))

(define-syntax-rule (@implv sym)
  (-> (@ '(language python3 impl) 'sym)))

(define-syntax-rule (@impl sym arg ...)
  (-> (call (@implv sym) arg ...)))

(define (do-assign targets val env toplevel)
  (define (doit id val)
    (if toplevel
        '(define ,id ,val)
        (let ((sym (lookup id env)))
          '(set! ,(if sym '(lexical ,id ,sym) '(toplevel ,id)) ,val))))
  (let ((ids (get-targets targets)))
    (if (null? (cdr ids))
        (doit (car ids) val))))
```

- `@impl` är ett makro. För att se hur det fungerar kan man se på följande anrop i Guiles REPL:

  ```
  scheme@(guile-user)> (@impl foo (+ 1 2))
  $1 = (call (@ (language python3 impl) foo) 3)
  ```

  Det som har skett är helt enkelt att ett Tree-IL uttryck med ett anrop till funktionen `foo` i modulen `(language python3 impl)` med argumentet `3` [18] har genererats. Modulen i fråga är den modul som bland annat innehåller många av de inbyggda Python 3 funktioner som har implementerats.

- `do-assign` ser till att värden antingen definieras eller skrivs över i ett lokalt scope beroende på om uttrycket som kompileras är ett toppnivåuttryck eller inte. För tillfället hanterar denna funktion endast fallet när `targets` är en lista med ett element.

3.3 Objekt

Som förklarades i sektion 2.3 representeras all data i Python på något sätt med hjälp av objekt. [Foud] Vad detta innebär förklaras lättast med ett exempel:

```
>>> (1).__class__
<class 'int'>
```

`__class__` är ett specialattribut vars värde är klassen som ett objekt tillhör. För just detta exempel visar det sig att heltalet 1 representeras i Python av ett objekt av typen *int*.

[18] 3 kommer från värdet på uttrycket `(+ 1 2)`.

Vid kompilering av objekt valde vi att översätta dessa till Guiles officiella objektsystem, GOOPS. Det finns dessvärre stora skillnader mellan Pythons och GOOPS syn på objektorientering. Några av dessa beskrivs i avsnitt 4.2.1.

Vår implementation av klasser försöker efterlikna den som hittas i Python så mycket som möjligt. Vi använder till exempel inte `define-method`, som definierar en generisk metod i GOOPS, för metoder som finns i klasser vi kompilerar. Dessa återfinns i stället som attribut[19] för klassen.

I funktionen `comp` som förklaras ovan hittar vi följande kod:

```
...
((<class-def> ,id ,bases ,keywords ,starargs ,kwargs ,body ,decos)
 (do-assign id
            (comp-class-def id bases keywords starargs kwargs body decos env)
            env toplevel))
...
```

Klassdefinition, `<class-def>`, mönstermatchas och ansvaret för kompileringen av klassen skickas vidare till `comp-class-def`. Denna funktion går igenom argumentet `body` (en lista av statements) och särskiljer de uttryck som senare kommer bli attribut i klassen från andra uttryck. Denna lista används senare i ett anrop till `make-python3-class`:

[19] Nästa sektion går igenom attribut och hur de är implementerade.

```
(define* (make-python3-class name bases body
                             #:key (keywords #f) (starargs #f)
                                   (kwargs #f)   (decos #f))
  "Creates a Python 3 class called NAME. BASES is a list of base class
instances. BODY is an alist containing symbols mapped to values. If the
symbol used is #f just evaluate the value. Otherwise bind the value to
the class's standard '__dict__' attribute. This function returns a
callable python 3 class."
  (let* ((sym (gensym (string-append (symbol->string name) "$")))
         (class-name (string->symbol (string-concatenate
                                      `("<" ,(symbol->string sym) ">")))))
    (eval
     `(begin
        (define-class ,class-name (<py3-object>)))
     (resolve-module '(language python3 impl)))
    (let ((class
           (make (@@ (language python3 impl) <py3-type>)
             #:d ((@@ (language python3 impl) make-attrs)
                  `((__bases__ . (,(@@ (language python3 impl) py3-object))))))))
      (slot-set! class 'procedure
                 (lambda ()
                   (let ((obj (make (module-ref (resolve-module
                                                 '(language python3 impl))
                                                class-name)
                                    #:d (make-hash-table 7))))
                     (setattr obj '__class__ class)
                     (slot-set! obj 'procedure
                                (lambda (. rest)
                                  (apply (getattr obj '__call__) rest)))
                     obj)))
      (map (lambda (x)
             (if (car x)
                 (setattr class (car x) (cdr x))
                 (cdr x)))
           body)
      class)))

(define-class <py3-object> (<applicable-struct>)
  (id #:getter py-id #:init-form (gensym "pyclass$"))
  (type #:getter py-type #:init-keyword #:t)
  (dict #:getter py-dict #:init-keyword #:d))

(define-class <py3-type> (<py3-object>))
```

Anta att en användare vill definiera en ny klass `Foo`. Det första som sker är att ett unikt klassnamn, exempelvis `<Foo$001>`, genereras och binds till `class-name`. Detta namn används till den klass i GOOPS som senare kommer vara basklass till alla objekt av Pythonklassen `Foo`. Identifieraren `Foo` binds sedan till ett objekt av typen `<py3-type>`. Då `<py3-type>` och `<py3-object>` är en

subklass till typen `<applicable-struct>` gör detta att instanser av dessa klasser är exekverbara i Guile. Den kod som körs när de exekveras beror på värdet i fältet `procedure`. I fallet för `Foo` binds `procedure` till en funktion vars uppgift är att skapa nya instanser av Pythonklassen `Foo` och initiera dessa. I initieringen ingår det bland annat att binda en funktion till det nya objektets `procedure` fält. Denna funktion skickar helt enkelt vidare alla sina argument till attributet `__call__`. [Foud, sektion 3.3.5] Det sista som sker i `make-python3-instance` är att allt som markerats som attribut i listan `body` blir ett attribut i `Foo`. Resterande värden exekveras.

Detta är långt ifrån en komplett implementation av objektsystemet i Python 3. En intresserad läsare kan till exempel exekvera `dir(object)` i en Python 3-REPL för att inspektera alla attribut som är definierade för `object`. Majoriteten av dessa attribut har för närvarande inget motsvarande attribut i vår implementation.

3.4 Attribut

I Python har objektinstanser ett antal attribut associerade med sig.[20] Detta saknar dock GOOPS-objekt, det som ligger närmast i GOOPS är fält.[21] Den största skillnaden mellan dessa två koncept är att de fält en instans har oftast kan bestämmas vid kompilering. Om man har två olika instanser av samma klass så har dessutom båda varsin uppsättning av likadana fält. Vad gäller attribut i Python så är de oftast dynamiska och beror mestadels endast på instansen och inte på klassen som definierade objekttypen. För att lättare kunna förklara vad som menas med dynamiska attribut presenteras följande exempel:

```
>>> class ClassA:
...     a = 10
>>> foo = ClassA()
>>> foo.a
10
>>> ClassA.a
10
>>> foo.a = 20
>>> ClassA.a
10
>>> foo.a
20
```

Som man kan se av exemplet så är attributet `a` för `foo` helt skilt från motsvarande attribut för `ClassA`, även fast `foo` kan läsa från `ClassA`s attribut om den inte definierar `a` själv. Om man vill att alla nya instanser av ett visst objekt ska ha ett eget attribut `a` kan man till exempel styra över detta genom att definiera specialfunktionen `__init__` för `ClassA`. Denna funktion kan då se till att varje ny instans definierar de attribut man är intresserad av. Det går även att lägga till nya attribut på flera andra sätt.

[20] Sedan Python 2.2 och med den nya typen av klasser så finns det även stöd för fält som liknar de typer av fält man pratar om i GOOPS.

[21] Vi använder från och med nu ordet fält när vi pratar om slots i GOOPS.

En konsekvens av detta dynamiska beteende när man hanterar attribut gör även att det inte går att bestämma var alla attribut definieras vid kompilering.

I Python finns det två inbyggda metoder som används för att leta upp och tilldela värden till attribut: `getattr` och `setattr`. Anropet `foo.a` är ekvivalent med anropet `getattr(foo, 'a')`. [Foub]

Detta är ungefär vad som sker vid kompileringen av attributuppslag i vår implementation.[22] När man stöter på ett `(<attribute> ,exp ,id ,ctx)` uttryck i kompileringsfasen produceras följande Tree-IL-kod:

```
(@impl getattr (comp exp e) `(const ,id))
```

I Python hittar man alla (modifierbara) attribut som en instans har tillgång till i attributet `__dict__`.[23] [Fouc] Genom att använda detta attribut för instanser av Pythonobjekt i Guile kan man implementera `getattr`:

```
(define-method (getattr (o <py3-object>) p)
  (if (slot-exists? o p)
      (slot-ref o p)
      (let ((h (hashq-get-handle (py-dict o) p)))
        (if h
            (cdr h)
            (let* ((type (cdr (hashq-get-handle (py-dict o) '__class__)))
                   (ret (lookupattr type p)))
              (if ret
                  (cdr ret)
                  (error (string-concatenate
                          `(,(object->string o) " has no attribute "
                            ,(symbol->string p)))))))))))

(define-method (lookupattr (o <py3-object>) p)
  (let ((h (hashq-get-handle (py-dict o) p)))
    (if h
        h
        (let lp ((classes (getattr o '__bases__)))
          (pmatch classes
            (()
             #f)
            ((,c . ,rest)
             (or (lookupattr c p) (lp rest)))))))))
```

[22] "Ungefär" syftar på detta exempel: `foo.a = 3`. Här vill man inte att ett anrop till `getattr` genereras utan `foo.a` ska fångas när man kompilerar tilldelningen och ett `setattr` genereras då i stället. I den nuvarande implementationen genereras dock bara `getattr` som det ska.

[23] `__dict__` är inte ett vanligt attribut i Python. `__dict__` har till exempel inte en referens till sig själv. Det är dock tillåtet att skriva över `__dict__` attributet och det är därför ett modifierbart attribut.

Som exempel kan man ta anropet `getattr(foo,'bar')`. Det första som sker är att man inspekterar om "bar" existerar som ett fält.[24] Värdet på detta fält returneras då omedelbart. Annars letar man i objektets dict (det vill säga i `__dict__` fältet) efter "bar" och returnerar värdet ifall det finns. Ifall även detta misslyckas skickas ansvaret att hitta rätt attribut till funktionen `lookupattr`. Man börjar med att leta igenom dicten för `foo`:s typ och sedan dicten för alla foo:s superklasser. Foo:s typ är här samma sak som värdet på `foo.__class__`. Superklasserna hittar man genom att göra en djupet-först-sökning på alla klasser man får av attributet `__bases__` om man börjar i `foo.__class__`.

Notering:

I en verklig implementation används inte en djupet-först-sökning. Detta då det kan bli tvetydigt vilka superklasser som ska besökas först i en såkallad diamantformad arvshierarki. [Foud, sektion: Custom Classes] I en riktig implementation av Python används *C3 Method Resolution Order algorithm* [Foua] för att hitta rätt ordning att traversera de olika klasserna. Då dokumentationen för MRO hittades sent och då semantiken inte påverkas alls för enklare klasshierarkier så lämnades definitionen med djupet-först-sökning som den var.

3.5 Funktionsanrop i Python

Inte bara typer inom Python är dynamiska. Även funktionsanrop sker på ett högst dynamiskt sätt. För att sätta sig in i problemet med att implementera ett funktionsanrop kan man se till följande Pythonfunktion:

```
def f(a,b):
    return (a,b)
```

Om man undantar namnade argument[25] och kwargs[26] kan man anropa ovanstående funktion på sju olika sätt med samma resultat:

`f(1), f(1,2), f(*[1]), f(*[1,2]), f(1,*[]), f(1,*[2])` och `f(1,2,*[])`

Om man även blandar in de båda uteslutna anropsätten finns det fler än 20 olika sätt att anropa ovanstående funktion med två argument på. Att översätta denna nivå av dynamik till Scheme är inte helt trivialt. För detta arbete valde vi att endast implementera de funktionsanrop som är giltiga då man inte tillåter namnade argument och kwargs.

Denna funktion är något mer komplicerad:

```
def g(a,b=2,*c):
    return (a,b,c)
```

[24] Tanken är icke muterbara attribut som till exempel `__class__` sparas som fält.
[25] `f(a=1)`
[26] `f(**{'a' : 1})`

Eftersom argumentet `c` fångar alla argument som inte binds till `a` eller `b` och lägger in dessa i ordning i en tupel kan den här funktionen anropas på ett oändligt antal olika giltiga sätt. Detta kallas för ett star-argument och markeras av stjärnan före `c` i funktionsdefinitionen.

I Scheme och Tree-IL finns det en motsvarighet till detta som kallas rest-argumentet. På samma sätt som att överblivna argument i Python binds till star-argumentet så binds även överblivna argument till rest-argumentet i Scheme. En skillnad är att man i Python även kan skicka med en lista av ytterliga argument förutom de vanliga argumenten. Vid ett funktionsanrop ska då de argument som inte binds till vanliga argument i listan konkateneras till listan av de vanliga argument (om det finns några) som blev över. Några exempelanrop med funktionen `g` ovan kan se ut så här:

```
>>> g(1,2,3,4,*[5,6,7])
(1,2,(3,4,5,6,7))
>>> g(1,*[5,6,7])
(1,5,(6,7))
```

Detta beteende har ingen motsvarighet i Scheme[27] och behöver därför hanteras separat. Ett sätt är att översätta alla sorters funktioner i Python till en funktion som tar två argument, nämligen rest-argumentet och ett så kallat keyword-argument. Ett keyword-argument är motsvarigheten till ett namnat argument i Python. Keyword-argumentet är tänkt att fungera som ett star-argument vid ett funktionsanrop, det vill säga att det är bundet till en lista.[28] Själva kroppen i funktionsdefinitionen bäddas sedan in sedan av en funktion som tar ut alla argument ur `rest` och keyword-argumentet och binder sedan rätt värden till rätt indentifierare i metodkroppen. Exemplet med `g` ovan kan då se ut:

[27] Detta problem kan faktiskt lösas genom definiera funktioner med `define*` som tar frivilliga argument och sedan anropa dessa funktioner med `apply`. Däremot blir det då i stället svårare att implementera keywords och kwargs i python och det är en av anledningarna till att vi valde att lösa problemet på detta sätt.

[28] Keyword-argumentet motsvarar i kodexemplet nedan `args`.

```
(lambda-case
 ((() #f rest$001 (#f (#:args args$002 args$002))
   ((const #f)) (rest$001 args$002))
  (let-values
    (primcall apply (primitive values)
       (@impl fun-match-arguments
              (const g)
              (primcall list (const a) (const b) (const c))
              (const #t)
              (lexical rest$001 rest$001)
              (lexical args$002 args$002)
              (primcall list (const 10))))
    (lambda-case
      (((a b c) #f #f () () (a$004 b$005 c$006 locals$003))
       (primcall return (primcall list (lexical a a$004)
                                       (lexical b b$005)
                                       (lexical c c$006))))))))
```

Här motsvarar det första lamba-case-fallet den yttre metoden som tar två argument, nämligen `rest$001` och `arg$002`.[29] Värdena på dessa argument skickas sedan vidare till metoden `fun-match-argument` som matchar värden som kommer in med argumenten som metoden har. Den gör även lite olika saker beroende på om det tredje argumentet som kommer in är sant eller falskt (om metoden har ett star-argument eller ej). De rätta värdena kommer sedan att skickas vidare via `let-values` [30] konstruktionen och bindas till argumenten `a`, `b` och `c` som är de argument som är definierade inne i Python-funktionen.

3.6 Namnbindning, moduler och scope

En fråga är hur namn binds till värden. Det finns avgörande skillnader mellan Schemes och Pythons sätt att hantera detta på, men även viktiga likheter.

Scheme och Python har det gemensamt att det finns precis ett namnrum som delas av funktioner och variabler. En funktion i Scheme är ett värde: ett lambda. Detta kan precis som andra värden sparas i en variabel och därmed också bindas till ett namn. Till skillnad från andra värden kan man anropa ett lambda. På ett liknande sätt är en funktion i Python också ett värde: ett funktionsobjekt. Ett anropningsbart objekt i Python är i princip ett objekt som bundit attributet `__call__` till ett lambda.[31]

Som vi konstaterat ovan finns det tre scope i Python. Det lokala scopet innehåller namn som på något sätt har definierats i det senaste[32] `def` eller `lambda` blocket. Vad detta "på något sätt" betyder kan enklast illustreras med ett exempel:

[29] `rest$001` och liknande namn är genererade namn.

[30] Dokumentation för `let-values` http://www.gnu.org/software/guile/manual/guile.html#SRFI_002d11

[31] eller ett annat anropningsbart objekt

[32] Även namn i tidigare `def` och `lambda` block kan räknas till det lokala scopet. Dock finns det vissa restriktioner i hur du får tilldela till variabler som inte är i det senaste blocket.

```
def f(b):
    if b:
        a = 5
    return a
```

I till exempel Scheme skulle motsvarande definition inte kompilera då variabeln `a` i if-satsen enbart hade varit definierat i scopet för just if-satsen. I Python kör denna kod om `b = True` och ger ett runtime-fel i övriga fall. För att översätta detta till Tree-IL behöver vi känna till alla variabler som potentiellt kan användas inne i till exempel en funktionsdefinition. Dessa variabler behöver sedan flyttas längst upp i funktionsdefinitionskroppen och bindas till ett "magiskt" värde som användaren inte själv kan tilldela variabeln. När en variabel senare tilldelas ett värde översätts detta alltid till ett anrop till `set!` i Tree-IL. För att lösa detta problem introducerar vi metoden `locals-and-globals`:

```
(define* (locals-and-globals s #:key (exclude '()))
  "This method returns the local and global variables used in a list of
statements. The returned value is on the form (LOCALS GLOBALS). The
EXCLUDE keyword argument is used to exclude certain symbols from the
returned local variables."
  (let lp ((stmts s) (locals '()) (globals '()))
    (pmatch stmts
      ((<global> ,vars) . ,rest)
       (lp rest locals (apply lset-adjoin eq? globals vars)))
      ((<assign> ((<name> ,var <store>)) ,val) . ,rest)
       (lp rest (lset-adjoin eq? locals var) globals))
      ((<if> ,test ,body ,orelse) . ,rest)
       (lp (append body orelse rest) locals globals))
      ((<function-def> ,id ,args ,body ,decos ,ret) . ,rest)
       (lp rest (lset-adjoin eq? locals id) globals))
      ((<class-def> ,id ,bases ,keywords ,starargs ,kwargs ,body ,decos) . ,rest)
       (lp rest (lset-adjoin eq? locals id) globals))
      ((,any . ,rest)
       (lp rest locals globals))
      (()
       (list (lset-difference eq? locals (append exclude globals)) globals)))))
```

Funktionen `locals-and-globals` letar upp alla namn som är definierade i till exempel en funktionskropp och returnerar två listor som motsvarar de lokala och de globala variablerna i kroppen. I denna implementation matchar `locals-and-globals` endast på triviala tilldelningar. Till exempel hittar den inte c:et i `(foo.a,c) = (1,2)`.

Den funktion som kompilerar `lambda-case`-uttrycket som omger funktionskroppen i Tree-IL[33] behöver uppdateras och behöver även kunna hantera alla lokala variabler som definieras inne i metoden. Dessa lokala variabler läggs till sist i listan med `REQ` argument till `lambda-case`-uttrycket. För funktionen `f` som definierades ovan kan det då se ut så här:

[33]Koden för detta gås igenom i avsnitt 3.5.

```
...
(lambda-case
  (((b locals$574 a) #f #f () () (b$573 locals$574 a$575))
   ...
```

Förutom `b` finns nu argumentet `a` och även ett helt nytt argument, `locals$574`. Det sistnämnda argumentet ska innehålla en lista med alla de lokala argument som kan definieras i detta block.[34]

En fråga är hur man hanterar moduler. Ett sätt är att göra som den befintliga Emacs LISP-implementationen. Det första man ska känna till är att Emacs LISP är ett Lisp 2-språk, det vill säga att man kan döpa exempelvis en variabel och en funktion till ett och samma namn utan att riskera kollisioner. Det finns dock inga moduler eller lexical scopes, utan bara globala namn. Av dessa skäl har man gömt undan alla variabler i en modul `(languages elisp runtime function-slot)` och på samma sätt för macron och variabler.

Detta kan vara olämpligt för ett språk med moduler och scoping. För vår del skulle det dessutom ge en lägre grad av integration än vad som önskas. Språkstödet skulle bli alltför separerat från övrig Guile-kod. Som tur är har Guile lexical bindings och stöd för moduler. Man kan i stället översätta Python-modulen `foo.bar.a` till Guile-modulen `(foo bar a)`. Om man dessutom bygger Python-funktioner med mera på ett sådant sätt att man kan använda dem direkt från exempelvis Scheme, ger detta en hög nivå av integration.

När man startar Guiles REPL hamnar man automatiskt i modulen `guile-user`. Analogt med detta körs Pythons huvudprogram alltid i modulen `__main__`. Det framstår som lämpligt att den senare ska fungera som ett alias för den förra.

Moduler kan importera andra moduler, och i detta fall få tillgång till de namn som moduler deklarerat som publika. I Guiles REPL kan man interaktivt byta module eller importera andra moduler:

```
scheme@(guile-user)> ,m (test)
scheme@(test)> (use-modules (ice-9 popen))
scheme@(test)>
```

3.7 Operatorer

Grundläggande operatorer har implementerats genom en ganska rak översättning:

```
(<add> expr1 expr2 env)
```

Till någonting som i Scheme skulle bli:

```
(+ (compile expr1 env) (compile expr2 env))
```

[34] Detta argument används för topnivå-funktionen `locals()`, se http://docs.python.org/py3k/library/functions.html?highlight=locals#locals.

Eftersom + är en `defgeneric` kommer vi kunna ange våra egna definitioner för egna typer.

3.8 Flödeskontroll

För tillfället är if-satser den enda typen av flödeskontroll som implementerats. Nedan följer ett exempel på en if-sats i Python 3:

```
if a:   pass
elif b: pass
else:   pass
```

Vår parser producerar följande AST för detta exempel:

```
(<module> ((<if> (<name> a <load>)
                 (<pass>)
                 ((<if> (<name> b <load>)
                        (<pass>)
                        (<pass>))))))
```

Detta kompileras sedan till en Tree-IL-representation liknande denna:

```
(if (toplevel a)
    (void)
    (if (toplevel b) (void) (void)))
```

I avsnitt 2.5 diskuteras vilka värden i Python som har ett sant sanningsvärde. Kortfattat kan man säga att detta bestäms antingen via fältet `__bool__` eller `__len__`. I Tree-IL och Scheme däremot anses ett värde vara sant i en if-sats om den inte evaluerar till `f` eller `nil`.[35] [Guib] Detta är därför inte en helt korrekt implementation av if-satser.

3.9 Testning

För att hjälpa till under implementationen utvecklades ett enklare domänspecifikt språk för testning. Vi utvecklade även en testprogram som läser in alla testfiler i en mapp för att sedan körs testerna. Vilket sorts test som körs beror på prefixet på filnamnet som testet ligger i. Till exempel så evalueras alla tester som ligger i filer med prefixet `eval-`. Strukturen på själva språket ses lättast med ett exempel:

[35] Att även `nil` anses vara falskt är inte dokumenterat. Det är dock lätt att testa detta genom att evaluera `(if nil 1 2) ==> 2`.

```
NAME=Global return
EXPECTED=2
### START CODE
a = 2
def f():
    return a
f()
### END CODE

NAME=Toplevel list from list assign
EXPECTED=9
### START CODE
[a,b] = [5,4]
a+b
### END CODE
```

Dessa testfall är tagna från en fil med tester som ska evalueras. Det förväntade värdet för en given evaluering hittas efter `EXPECTED`. Även detta värde evalueras och ska därför vara ett giltigt Scheme-uttryck. Det hade till exempel varit tillåtet att i det sista testfallet att byta ut `9` mot `(+ 5 4)`

3.10 Diskussion

Vad gällande parsningen av Python gick vi in i projektet med övertygelse om att PEG är den teknik som var mest lämplig. Ovan har några av fördelarna med PEG i jämförelse med CFG beskrivits. Även om Guile-modulen är experimentell och relativt otestad så verkar den på väg i rätt riktning.

Preprocessorn vi skrev kom aldrig till användning då det skulle kräva en hel parser och vi använde i stället Pythons egna lexer och parser. För att göra det så enkelt som möjligt att parsa python filer är det dock viktigt att ha en liknande funktionalitet.

Hur noga det är att Python objekt interagerar väl i Scheme-miljö är något vi diskuterat i flera omgångar med utgångspunkt i de språk som Guile redan har stöd för, ECMAScript och Emacs LISP. ECMAScript integrerar väl med Scheme så att det exempelvis är möjligt att definiera en funktion i ECMAScript och sedan anropa den från Scheme. Med Emacs LISP finns det ingen sådan interaktion. Vi menar att det är lämpligt och möjligt att sträva efter en hög nivå av integration, även om det svårligen kan fås att bli helt analogt. Vissa element av det ser vi dock som viktigare än andra, till exempel att det ska vara fullt möjligt att anropa ett Python-objekt som implementerar `__call__` precis som om det vore en “vanlig” funktion även i Scheme.

3.11 Slutsatser

Även om Python har vissa skillnader mot Guile när det gäller scope, variabel-hantering, objekt-attribut och moduler så tyder våra efterforskningar på att det är fullt möjligt att implementera

fullt stöd för Python-liknande syntax i Guile även om vissa inskränkningar kan komma att behöva göras för att förenkla koden eller av prestandaskäl.

Med hjälp av en preprocessor, sådan som vi gjort, och PEG är vi övertygade om att det är fullt möjligt att tolka Python. För att sedan översätta det till något Guile förstår så kommer det även vara fullt möjligt att implementera funktionsanrop och enkla operationer. För att implementera objektsystemet, som allt i Python är beroende av, så krävs dock ett mer kraftfullt system än enkla GOOPS-objekt kan erbjuda. Med hjälp av MOP som presenteras i nästa kapitel skulle det vara fullt möjligt att implementera ett system som liknar Pythons objektsystem. På så sätt skulle vi kunna utveckla ett system som även tillåter objekt skapade i Guiles Python 3-läge att användas i Scheme-läget på ett intuitivt sätt.

4 Metaobjektprotokollet (MOP)

Under arbetets gång blev vi i gruppen mer och mer uppmärksammade på metaobjektprotokollet (MOP). Vi blev inspirerade av hur det i Common Lisp Object System (CLOS)[36] kunde ersätta flera olika implementationer av objektorientering, alla med olika regler för till exempel arv och instansiering, samtidigt som övergången kunde göras smärtfritt och tillåta interaktion mellan de olika systemen för de projekt som var beroende av dessa tidigare system.[37] MOP är även intressant för vårt ändamål då det ger oss en helt ny infallsvinkel på hur man på ett bra sätt integrerar Python 3 i GNU/Guile. I detta kapitel vill vi därför göra följande:

- Berätta om hur metaobjektprotokollet uppstod och vilka problem det löste.
- Presentera några exempel på användning av metaobjekt för att ge läsaren en förståelse för dem.
- Diskutera möjligheten av en komplett implementering av semantiken i Pythons objektsystem i metaobjektprotokollet. Denna implementation föreslås senare användas som grund för implementationen av resterande semantik i Python 3.

4.1 Historia

> "In fact the very notion of a single standardized object-oriented extension to Common Lisp was an inherently incompatible change, since the set of earlier extensions were not compatible among themselves." [KR91, sid 7]

Detta citat beskriver tydligt problemet som skaparna av CLOS stod inför. Det fanns redan flera välanvända system för objektorientering i Common Lisp och alla hade olika tankar om hur till exempel arv, klassrelationer och metodanrop skulle hanteras. Trots att alla dessa system hade samma grundidé så skilde de sig så pass mycket åt på detaljnivån att interkompabilitet mellan dem verkade ouppnåeligt. Ett renodlat eller traditionell system för objektorientering var helt enkelt inte tillräckligt uttrycksfullt för att kunna fungera som ersättning. Det var dessa behov som drev på utformningen av MOP.

Användarna av CLOS behövde kunna styra hur arv prioriteras, hur tillgången till fält går till och ville dessutom kunna göra saker som med sina objekt som normalt inte faller under kategorin normal objektorientering. [KR91, kap 3.7.1] Att öppna upp systemet helt och hållet var aldrig ett alternativ då även prestanda och användarvänlighet var två viktiga mål i CLOS-projektet. Det som behövdes var något som tillät användarna att ändra på specifika egenskaper i sina objekt samtidigt som resterande egenskaper fortfarande fungerade förutsägbart. Detta är något som objektorientering redan är bra på att beskriva. Genom att skriva om precis de delar som användaren är intresserad att ändra i termer av generiska metoder blev det möjligt att lägga till eller ändra funktionalitet samtidigt som all annan funktionalitet bevarades.

[36] Se vidare https://secure.wikimedia.org/wikipedia/en/wiki/Common_Lisp_Object_System.
[37] Det var dels på grund av MOP som CLOS senare blev det officiella objektsystemet för Common Lisp.

Utvecklarna av CLOS valde att implementeras detta[38] med hjälp av de fem metaobjektklasserna `class`, `slot-definition`, `generic-function`, `method` och `method-combination`. [KR91, sid 137] Var och en med en standardimplementation `standard-class`, `standard-slot-definition` etc. Syftet med dessa klasser är att ge en standarddefinition av de flesta delar som en användare kan tänka sig ändra. Genom att skapa en ny klass som är en subklass till någon av metaobjektklasserna skapar man en ny metaobjektklass.

4.2 Att använda MOP

Som namnet antyder är MOP (metaobjektprotokoll) ett objektorienterat system för att specificera hur ett objektorienterat system ska fungera. I grunden finns en uppsättning fördefinierade beteenden – ett protokoll – som sedan kan överlagras och på så sätt förändra hur klasser eller objekt som ärver från de klasserna beter sig. Detta gör det möjligt för ett MOP-baserat objektorienterat system att emulera många olika andra objektorienterade system alltefter programmerarens behov. Man kan till och med använda flera olika system samtidigt.

Till exempel är det möjligt att skapa en metaklass vars klasser vid arv ändrar prioriteringsordningen bland förälderklasserna. I GOOPS, som är den populäraste implementation av ett MOP i Guile, görs detta genom genom att skriva en egen implementation för den generiska metoden `compute-cpl` för den metaklass det gäller. Varje instansiering av denna klass, och klasser som ärver denna, kommer då, i enlighet med protokollet, anropa `initialize` som i sin tur anropar `compute-cpl` som, utifrån det blivande objektets klass, räknar ut att det är den nya versionen av `compute-cpl` som ska användas i stället för den ursprungliga `compute-std-cpl`.

4.2.1 Objektorientering i Guile med GOOPS

För den som är van vid C-liknande objektorienterade programmeringsspråk kan objektorienteringen GOOPS använder sig av, te sig konstig och bakvänd. Till exempel är metoder inte attribut på objektet på samma sätt som till exempel i Python där ett metodanrop skulle skrivas som `foo.bar(x, y)`. I stället används generiska metoder som tar objekten som argument och därifrån räknar ut vilken version av funktionen som ska anropas. Alltså skulle samma anrop i GOOPS bli `(bar foo x y)` och den generiska metoden `bar` skulle använda sig av `foo`'s klass-struktur för att beräkna vilken implementation av `bar` som faktiskt ska anropas. Detta innebär bland annat att det inte finns någon speciell syntax i Scheme för att hantera klasser och objekt utan hela språkets kraft kan användas för objektorienteringen likväl som för funktionell programmering eller andra paradigm som stöds av Scheme.

[38] Egentligen är detta hur det implementeras i Clossete som är det referensspråk som används i *The art of the Metaobject Protocol*. [KR91] Både implementation av CLOS och Schemes motsvarighet, GOOPS, är dock näraliggande implementationer av systemet i Clossete så det mesta kan översättas direkt emellan de olika systemen.

4.3 Python i MOP

I detta avsnitt presenterar vi hur delar av Pythons objektsystem kan implementeras i GOOPS med hjälp av metaobjekt. Först visas en enkel metaklass som förändrar hur man hittar ett fält i GOOPS. Denna klass utökas därefter med funktionalitet för arvshantering. Till sist läggs det till stöd för det som i Guile kallas för applicerbara structar.

4.3.1 Attribut

I avsnitt 3.4 finns en kort genomgång om hur man letar attribut i Python. En viktig lärdom som man kan dra av detta avsnitt är att den klass som ett visst attribut finns inte kan bestämmas vid kompileringen. GOOPS standardklasser har ingen motsvarighet till detta dynamiska beteende. När man skapar ett nytt objekt räknas det ut exakt vilka fält denna instans har tillgång till och var dessa fält är definierade. En fördel med detta är att det blir mycket enklare att implementera en effektiv algoritm för att hämta värdet på ett fält. Då informationen om var ett visst attribut är definierat inte räknas ut dynamiskt i GOOPS ser vi två sätt att lösa detta:

- Implementera det som ovan och helt enkelt hämta spara attribut i varje instans privata dict och sedan hämta ut dessa värden med hjälp av `getattr`. I språkmiljön för Python är detta inget problem då ett anrop till `foo.bar` automatiskt översätts till `getattr(foo,'bar')`. Om man vill komma åt `foo.bar` i någon annan språkmiljö än just Python krävs det att man på något sätt importerar funktionen `getattr`.
- En annan möjlighet är att skapa ett metaobjekt som utöver funktionaliteten för vanliga fält även har stöd för attribut som liknar de som finns i Python. Om detta görs på rätt sätt medför detta att Pythons objektorientering integreras i GOOPS objektsystem i stället för att bli en självständig utökning av det.

I CLOS finns det en viktig generisk metod som anropas då en användare letar efter ett visst fält, `slot-value-using-class`. Om man definierar en ny metaklass som skriver över denna metod kan man styra över hur man letar efter fält för just denna metaklass. I fallet med Pythonobjekt kan man helt enkelt låta metoden `slot-value-using-class` vara ett alias för `getattr`.

```
(define-class <py3-object-class> (<class>))

(define-class <py3-object> ()
  #:metaclass <py3-object-class>)

(define-method (slot-ref-using-class (object <py3-object-class>) slot
  (getattr object slot))
```

Här används metoden `slot-ref-using-class` i stället för `slot-value-using-class` som är den metod i GOOPS som ligger närmast motsvarande metod i CLOS. Tyvärr fungerar inte denna lösning då

Funktionen `slot-ref-using-class` inte är generisk i GOOPS. Trots vissa försök att kommunicera med Guile-communityt har vi inte kunnat ta reda på varför detta är fallet.[39] Det bästa sättet att lösa detta på är helt enkelt att ändra på implementationen av `slot-ref-using-class` till en generisk metod.[40] För närvarande är denna funktion implementerad i ett bibliotek till Guile skrivet i C. Att föreslå och genomföra en sådan ändring bör även öka intresset bland Guile-communityt. Om det finns en anledning till att denna funktion inte är generisk bör detta komma fram då.

En annan metod som är värd att undersöka närmare är `allocate-instance`. I GOOPS anropas denna metod då en ny instans av ett objekt behöver allokeras. I sektion 3.3 presenterar vi hur vi skapar Pythonklasser i Guile med hjälp av funktionen `make-python3-class`. Förutom att skapa en klass ser även denna funktion till att ett anrop till klassen skapar en ny, korrekt initierad instans. Med korrekt menas här att fält såsom `procedure` och `__dict__` initieras till rätt värden. Att denna funktionalitet ligger i `make-python3-class` är inte helt lämpligt. Om en användare till exempel vill skapa instanser av Pythonobjekt utanför Python 3 språkmiljön kan de inte använda sig av standardfunktionen i GOOPS för detta, `make`. Det som behöver göras är att lyfta ut mycket av funktionaliteten från `make-python3-class` och flytta det till den tidigare nämnda generiska metoden `allocate-instance`.

4.3.2 Arv

I avsnitt 3.4 nämns kortfattat *The C3 Method Resolution Order algorithm* (MRO). Givet en klass räknar denna algoritm ut i vilken ordning superklasser ska besökas när en klassmetod exekveras. Denna algoritm används även för att bestämma i vilken ordning som klasser gås igenom för att hitta attribut. [Foua] Varje gång en ny klass definieras körs denna funktion och resultatet sparas i attributet `__mro__` för klassen. Vår implementation använder för tillfället en djupet-först-sökning som behöver köras varje gång ett attribut eller metod behöver lokaliseras för ett objekt. I en framtida implementering krävs det dock att MRO används i stället.

Det första som behöver göras är helt enkelt att implementera MRO-algoritmen i exempelvis Scheme. Denna funktion körs därefter varje gång en ny klass definieras för att räkna ut arvshierarkin för den nya klassen. I vissa fall finns det ingen giltig ordning. CPython slänger i dessa lägen ett `TypeError`:

[39] http://article.gmane.org/gmane.lisp.guile.devel/14393

[40] En annan möjlig lösning kan vara exempel definiera `slot-missing`. Denna generiska metod körs av ett antal funktioner då ett fält inte kan hittas. Denna lösning kringgår dock bara problemet utan att ta hänsyn till konsekvenserna.

```
>>> class A: pass
>>> class B(A): pass
>>> class C(A,B): pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Cannot create a consistent method resolution
order (MRO) for bases B, A
```

Då vi föreslår en komplett implementering av Pythons objektsystem i GOOPS med hjälp av metaobjektprotokollet behövs det en generisk metod för att styra vad som sker vid klassinstansiering. I Scheme används funktionen `define-class` när en ny klass definieras. Denna funktion är inte generisk men använder sig i stället av en annan generisk metod: `make`. [Guia] Målet är att MRO-algoritmen körs varje gång men definierar en ny Python-klass oavsett språkmiljö. På grund av detta räcker det inte längre att enbart specialisera `make` för nya instanser av `py3-object-class` [41] då detta objekt har `<class>` som superklass. Det krävs alltså åtminstone en till metaklass för att kunna styra definieringen med klasser. Nedan följer en utveckling av implementationen i 4.3.1 med stöd för MRO och fältet `__dict__`.

```
(define-class <py3-meta-meta-class> (<class>))

(define-method (make (class <py3-meta-meta-class>) . rest)
  (let ((obj (next-method)))
    (slot-set! obj '__mro__  (mro obj))
    (slot-set! obj '__dict__ (make-hash-table 7))
    obj))

(define-class <py3-meta-class> (<class>)
  __mro__ __dict__
  #:metaclass <py3-meta-meta-class>)

(define-method (make (class <py3-meta-class>) . rest)
  (let ((obj (next-method)))
    (slot-set! obj '__dict__ (make-hash-table 7))
    obj))

(define-class <py3-object> ()
  __dict__
  #:metaclass <py3-meta-class>)

(define test-object (make <py3-object>))
```

Här finns två definitioner av `make`. Den första används då en ny Python-klass definieras medan den andra är den som körs då en nytt objekt instansieras. Som förklarades ovan är `__dict__` här definierat som ett

[41] `<py3-object-class>` definierades i förra avsnittet.

Givet en definition av funktionen `mro` kan detta testas i Guiles REPL:

```
scheme@(guile-user)> (slot-ref <py3-object> '__dict__)
$1 = #<hash-table 0/31>
scheme@(guile-user)> (slot-ref <py3-object> '__mro__)
$2 = (<py3-object>)
scheme@(guile-user)> (slot-ref test-object '__dict__)
$3 = #<hash-table 0/31>
scheme@(guile-user)> (define-class test (<py3-object>))
scheme@(guile-user)> (slot-ref test '__dict__)
$4 = #<hash-table 0/31>
```

De tre sista raderna ger ett exempel på hur en användare fritt kan använda GOOPS egna syntax för att skapa egna Pythonklasser.

4.3.3 Applicerbara strukturer

Eftersom alla Python-objekt som implementerar `__call__` kan anropas som om de var funktioner anser vi att detsamma måste gälla för Python-objekten också i Scheme-miljön. Alla funktioner i Python kan ses som objekt som har ett lambda tilldelat till `__call__`. Detta attribut kan även ändras under programmets exekvering.

För att kunna använda ett GOOPS-objekt som om det vore en funktion, att kunna applicera objektet, måste detta ärva från klassen `<applicable-struct>` som har en fältet `procedure`.

```
scheme@(guile-user)> (define o (make <applicable-struct>))
scheme@(guile-user)> (slot-set! o 'procedure (lambda (x) (+1 x)))
scheme@(guile-user)> (o 2)
3
```

En intuitiv lösning skulle vara att ändra klassen på objektet om `__call__` tilldelas ett lambda. Detta skulle dock leda till att det nya objektet får en annan identitet än det hade innan vilket inte stämmer överens med hur likhet mellan objekt fungerar i Python. Detta betyder alltså att varje GOOPS-objekt som representerar Python objekt måste ärva `<applicable-struct>` och implementera fältet `procedure` oavsett om `__call__` är satt eller inte. Fältet `procedure` ska då peka på en funktion som kollar om objektet har attributen `__call__` satt och då anropa funktionen som finns där eller som rapporterar felmeddelande om användare försöker applicera ett objekt som inte har attributet satt. I funktionen `make-python3-class` som visades i avsnittet 3.3 finns en lösning till detta:

```
(define* (make-python3-class name bases body)
     ...
     (slot-set! obj 'procedure
                (lambda (. rest)
                  (apply (getattr obj '__call__) rest))) ... )
```

Varje gång ett nytt objekt instansieras i Pythons språkmiljö körs koden ovan som binder fältet `'procedure` för `obj` till en anonym funktion med ett argument, `rest`. Detta argument kommer bindas till en lista innehållande alla argument för ett givet anrop. Då `(getattr obj '__call__)` i sin tur returnerar den funktion vi vill applicera alla dessa argument på. Om attributet inte finns eller om det är bundet till ett icke applicerbart värde kommer Guile att kasta ett fel.

Ansvaret för denna funktionalitet bör flyttas över till metaobjektprotokollet. Lämpligtvis till den generiska funktionen `make` som definierades i förra avsnittet. Den uppdaterade metoden kommer då att se ut:

```
(define-method (make (class <py3-meta-class>) . rest)
  (let ((obj (next-method)))
    (slot-set! obj '__dict__ (make-hash-table 7))
    (slot-set! obj 'procedure
               (lambda (. rest) (apply (getattr obj '__call__) rest)))
    obj))
```

Förutom denna ändring krävs det även att basklassen `<py3-object>` uppdateras så att denna ärver från `<applicable-struct>`. Även metaobjektet `<py3-meta-class>` behöver ändras på ett liknande sätt.

4.4 Diskussion

Då en MOP-implementation i GOOPS kan ses som mer generell än Pythons objektsystem bör det vara fullt möjligt att begränsa GOOPS och forma det för att, på så sätt, få ett system som agerar som Pythons objektsystem. Potentiella problem skulle kunna vara att implementera Pythons metaklasser på det sättet och att garantera ett intutivt gränssnitt för objekten i Scheme miljö. Men då metaklasser kan ses som vilket annat attribut som helst till Pythons klasser så är det möjligt att det inte är ett problem överhuvudtaget.

Applicerbara strukturer är inte något som ses som viktigt i Scheme-världen, men för att implementera Python med god integration med Scheme krävs denna funktionalitet. Guile kräver då som sagt att objektet ärver från `<applicable-struct>`. En ordentlig implementation av en metaklass för Python behöver, som vi har sätt i detta kapitel, ärva från flera metaklasser. Så som `<applicable-struct>` är implementerat just nu betraktes inte objekt som ärver av någon mer metaklass än `<applicable-struct-class>` [42] som ett applicerbart objekt. Detta leder till allvarliga begränsningar i implementationen. En potentiell lösning skulle vara att i stället se till att användaren måste använda en funktion så som `(py-call pyobj)` för att applicera ett Python objekt. Detta skulle dock kunna leda till stor förvirring ur integrationssynpunkt för användarna.

[42] `<applicable-struct-class>` är metaobjektet för `<applicable-struct>`. Följande klass kommer automatiskt att vara av denna metaklass: `(define-class foo (<applicable-struct>))`.

4.5 Slutsatser

MOP är ett kraftfullt och mångfaceterat verktyg, särskilt när det gäller implementering av ett språk som Python, där allt representeras som objekt.[Foud] De största problemen vi stött på när det gäller GOOPS är begränsningar i implementationen. Exempelvis är problemet med applicerbara strukturer väldigt problematiskt och antagligen resultatet av en felaktig implementation.

Nyckeln till att implementera Python, specifikt i Guile men även generellt, är att få objektsystemet att fungera bra. MOP är ett stabilt och väldefinierat system för att formulera objektsystemet. Hade vi känt till MOP från början hade vårt inledande arbete antagligen tagit en helt annan riktning.

Trots att det blev vissa problem under implementeringen i detta kapitel visade det sig att MOP lämpar sig väldigt väl för att beskriva objektsystemet i Python. Med korta och eleganta exempel lyckades vi på ett bra sätt fånga delar av Pythons objektsemantik.

5 Diskussion

Meningen var från början att vårt projekt skulle implementera en rimligt stor delmängd av Python. Formuleringen "rimligt stor" valdes medvetet därför att den är flexibel. En stor del av arbetet innebar nämligen att undersöka vad som skulle göras och hur, snarare än att faktiskt implementera.

Vi hade från början som ambition att även bygga en test suite för vår implementation men detta skedde inte i någon större utsträckning på grund av att vi inte kom så långt i implementationen. För att göra bra sådana tester hade det dock kunnat vara intressant att kolla på test suiter från andra projekt så som PyPy, Python implementerat i Python, eller andra implementationer av Python.

Det fanns många variabler som i början av arbetet inte med lätthet kunde uppskattas. På grund av denna svårighet i att göra en preliminär bedömning av vad som var realistiskt att förvänta sig av slutprodukten, hamnade fokus på att dokumentera de målsättningar som satts upp och vilka begränsningar som framträtt under arbetets gång.

Det första vi måste konstatera är att projektet inte kom i närheten av att uppnå de mål vi satte upp: ingen av de uppsatta milstolparna nåddes. I efterhand framstår detta inte som alltför märkligt, då vi från början inte insåg hur stort språk Python 3 faktiskt är, eller hur stor del av det som är beroende på dess bibliotek. Trots att syntaxen är relativt liten, är språket enormt. Redan för mycket små program kräver dessutom som vi visat ett avsevärt språkstöd.

Den stora, och verkligt svåra biten, är objektorientering. Som vi visat är detta i själva verket en av de saker ett projekt som detta måste ta itu med innan något annat kan implementeras. Den tidsplan som lades upp utgick tvärtom från det motsatta antagandet: att det skulle vara någonting som vi kunde komma till först senare. Detta berodde på en felaktig förståelse av hur Python fungerar, och ett illa underbyggt antagande att det skulle fungera ungefär som Java.

En sak som är lätt att förledas av är att, Python Language Reference, är en liten och kompakt text. Läsare av dokumentationen måste dock komma ihåg att språket i själva verket är inkomplett utan implementation av stora delar av standardbiblioteket. Med denna ihopblandning av språkimplementation och standardbibliotek krävs en tydligare dokumentation av detta än vad vi anser att den officiella dokumentationen erbjuder.

Projektet har vidare dragits med svårigheten att vi flera gånger varit tvungna att ta avsevärd tid enbart för att förstå hur språket fungerar. Detta kan dels bero på deltagarnas låga kunskapsnivå vid ingången, men måste även delvis förklaras med att Python 3 är komplext med dokumentation fokuserad på användande snarare än implementation. Specifikationen av vissa språkkonstruktioner har många gånger varit otillfredsställande. Många gånger verkar den snarast rikta sig till någon som redan är expert på Python 3-implementationer medan andra delar av den är grundläggande och explicita.

I vårt projekt har vi valt att minimera arbetet gällande de tidigare stegen i kompileringsprocessen, nämligen lexning och parsning. I detta syfte valde vi det upplägg som beskrivits ovan, i avsnittet 3.1, där vi utnyttjat standardmodulen `ast`. Detta hade flera fördelar, varav den främsta är att vi

snabbt kom igång med implementering av kodgenerering. Dessutom skulle nya förändringar i språket direkt kunna parsas och endast semantiken behöva kompletteras. Detta kan utnyttjas i framtiden för att uppdatera projektet till nya versioner av Python och förhoppningsvis göra det möjligt att snabbt uppdatera Guiles Python 3-implementation till senaste versionen av Python 3.

Det finns dock vissa problem med detta val. Ett är att en lokal Python 3-installation krävs för att kompilering ska kunna ske, och att denna dessutom måste vara av rätt version för att frontendet ska kunna tolka syntaxträdet. Detta gör att lösningen inte är speciellt robust eller lämplig för annat än utvecklingsändamål.

En bättre lösning verkade vara att skriva en helt ny parser baserad på den experimentella PEG-modulen. Detta var inte nödvändigtvis en god idé, eftersom vi stötte på fler än en bugg som hindrade arbetet. Dessutom lider modulen av att sakna aktiva utvecklare och vid rapportens skrivande har ingen utveckling av modulen gjorts sedan januari, fortsätter denna inaktivet kan andra alternativ vara värda att undersöka.

6 Slutsatser

Att implementera Python 3 är en komplicerad uppgift, och bör inte underskattas. Det är ett stort språk med många dolda detaljer, vilka gör det lättare för användaren men gör implementeringen betydligt mer komplex.

För att implementera Python på ett användbart sätt i Guile krävs ett bra objektsystem, och det mest realistiska sättet att göra detta på är att utnyttja GOOPS metaobjektprotokoll. Genom att integrera väl med MOP så får vi även möjlighet att utnyttja objekt från Python-miljön i Scheme utan att behöva konvertera emellan. För att detta ska vara möjligt krävs antagligen vissa förändringar i Guiles implementation av Scheme och dessutom behövs en, i Guile fullt integrerad parser.

En PEG-parser verkar vara en rimlig väg att gå då Python inte är helt kontextoberoende, förslagsvis kan dessutom den nya PEG-modulen i Guile användas när den bedöms stabil nog för att inkluderas i Guiles officiella distribution. Att basera arbetet på en experimentell branch framstår som olämpligt, mycket på grund av de problem vi stött på i våra tester.

A Ordlista

- **AST** – abstrakt syntaxträd. Resultatet av att parsa ett . En trädstruktur som motsvarar ett giltigt program.
- **Branch** – utvecklingsgren som är separerad från den huvudsakliga grenen. Meningen är att den ska kunna inkluderas i ett senare skede.
- **CFG** – Context-Free Grammar, som specificerar ett Context-Free Language. Det vanligaste sättet att uttrycka ett programmeringsspråks syntax.
- **GNU** – GNU's Not Unix. En rekursiv förkortning som är namnet på ett helt fritt operativsystem som utvecklas av Free Software Foundation.
- **GOOPS** – Objektsystemet i Scheme / Guile
- **Fält** – Slots i GOOPS
- **Namespace** – en behållare för namn. Inuti ett namespace riskerar namn inte att förväxlas med namn i ett annat namespace, även om de är identiska.
- **Parsa** – ungefär tolka. Det att konvertera en teckensträng till en AST (se ovan).
- **REPL** – Read-Eval-Print Loop. Interaktiv kodmiljö.

B Tree-IL-referens

(**lambda** META BODY)

- ...

(**lambda-case** ((REQ OPT REST KW INITS GENSYMS) BODY) [ALTERNATE])

- Ett möjligt anrop till ett lambda. `REQ` är en lista med de obligatoriska argumenten till detta lambda. `OPT` är frivilliga. `REST` är en atom eller `#f` om det inte ska finnas ett `REST` argument. `KW` är en lista med element av typen `(KEYWORD NAME GENSYM)`.

C Python 3 AST-definition

Följande PEG-liknande definition är den interna representationen av Python som används i implementeringen av Python till Tree-IL kompleringen.

Symbol	betyder	exempel
foo *	Lista av foo	Num* => (3 5) eller ()
foo ?	foo eller None	

```
-- behöver kanske ej annat än mod och interactive
Mod  <- (<module> Stmt*)                                ;; body
     /  (<interactive> Stmt*)                           ;; body

Stmt <- (<function-def> Ident                           ;; name
                        Arguments                       ;; args
                        Stmt*                           ;; body
                        Expr*                           ;; decorator_list
                        Expr?)                          ;; returns
     /  (<class-def> Ident                              ;; name
                     Expr*                              ;; bases
                     Keyword*                           ;; keywords
                     Expr?                              ;; starargs
                     Expr?                              ;; kwargs
                     Stmt*                              ;; body
                     Expr*)                             ;; decorator_list
     /  (<return> Expr?)                                ;; value
     /  (<delete> Expr*)                                ;; targets
     /  (<assign> Expr? Expr?)                          ;; targets value
     /  (<aug-assign> Expr Operator Expr)               ;; target operator value

     /  (<for> Expr Expr Stmt* Stmt*)                   ;; target iter body orelse
     /  (<while> Expr Stmt* Stmt*)                      ;; test body orelse
     /  (<if> Expr Stmt* Stmt*)                         ;; test body orelse
     /  (<with> Expr Expr? Stmt*)                       ;; context_expr
                                                        ;;   optional_vars body

     /  (<raise> Expr? Expr?)                           ;; exc cause
     /  (<try-except> Stmt* Except-handler* Stmt*)      ;; body handlers orelse
     /  (<try-finally> Stmt* Stmt*)                     ;; body finalbody
     /  (<assert> Expr Expr?)                           ;; test msg

     /  (<import> Alias*)                               ;; names
     /  (<import-from> Ident? Alias* Int?)              ;; module names level
```

```
     /  (<global> Ident*)                                ;; names
     /  (<nonlocal> Ident*)                              ;; names
     /  (<expr> Expr)                                    ;; value
     /  <pass>
     /  <break>
     /  <continue>


Expr <- (<bool-op> Boolop Expr*)                         ;; op values
     /  (<bin-op> Expr Operator Expr)                    ;; left op right
     /  (<unary-op> Unaryop Expr)                        ;; op operand
     /  (<lambda> Arguments Expr)                        ;; args body

     /  (<if-exp> Expr Expr Expr)                        ;; test body orelse
     /  (<dict> Expr* Expr*)                             ;; keys values
     /  (<set> Expr*)                                    ;; elts
     /  (<list-comp> Expr Comprehension*)                ;; elt generators
     /  (<set-comp> Expr Comprehension*)                 ;; elt generators
     /  (<dict-comp> Expr Expr Comprehension*)           ;; key value generators
     /  (<generator-expr> Expr Comprehension*)           ;; elt generators
     /  (<yield> Expr?)                                  ;; value
     /  (<compare> Expr Cmpop* Expr*)                    ;; left ops comparators
     /  (<call> Expr                                     ;; func
                Expr*                                    ;; args
                Keyword*                                 ;; keywords
                Expr?                                    ;; starargs
                Expr?)                                   ;; kwargs
     /  (<num> Num)                                      ;; n
     /  (<str> String)                                   ;; s
     /  (<bytes> String)                                 ;; s
     /  <ellipsis>

     /  (<attribute> Expr Ident Expr-context)            ;; value attr ctx
     /  (<subscript> Expr Slice Expr-context)            ;; value slice ctx
     /  (<starred> Expr Expr-context)                    ;; value ctx
     /  (<name> Ident Expr-context)                      ;; id ctx
     /  (<list> Expr* Expr-context)                      ;; elts ctx
     /  (<tuple> Expr* Expr-context)                     ;; elts ctx

Expr-context <- <load> / <store> / <del> / <aug-load>
             /  <aug-store> / <param>

Slice <- (<slice> Expr? Expr? Expr?)                     ;; lower upper step
```

```
      /  (<ext-slice> Slice*)                          ;; dims
      /  (<index> Expr)                                ;; value

Boolop <- <and> / <or>

Operator <- <add> / <sub> / <mult> / <div> / <mod> / <pow>
         /  <l-shift> / <r-shift> / <bit-or> / <bit-xor>
         /  <bit-and> / <floor-div>

Unaryop <- <invert> / <not> / <u-add> / <u-sub>

Cmpop <- <eq> / <not-eq> / <lt> / <lt-e> / <gt> / <gt-e>
      /  <is> / <is-not> / <in> / <not-in>

Comprehension <- (Expr Expr Expr*)                   ;; target iter ifs

Excepthandler <- (Expr? Ident? Stmt*)                ;; type name body

Arguments <- (Arg*                                   ;; args
              Ident?                                 ;; vararg
              Expr?                                  ;; varargannotation
              Arg*                                   ;; kwolyargs
              Ident?                                 ;; kwarg
              Expr?                                  ;; kwargannotation
              Expr*                                  ;; defaults
              Expr*)                                 ;; kw_defaults

Arg <- (Ident Expr?)                                 ;; arg annotation

Keyword <- (Ident Expr)                              ;; arg value

Alias <- (Ident Ident?)                              ;; name asname
```

Referenser

[ABB+98] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson, *Revised5 report on the algorithmic language scheme*, SIGPLAN Not. **33** (1998), no. 9, 26–76.

[For04] Bryan Ford, *Parsing expression grammars: a recognition-based syntactic foundation*, SIGPLAN Not. **39** (2004), no. 1, 111–122.

[Foua] Python Software Foundation, *The python 2.3 method resolution order*, `http://www.python.org/download/releases/2.3/mro/`, Hämtad 2012-05-12.

[Foub] _____, *Python library: Built-in functions*, `http://docs.python.org/py3k/library/functions.html`, Hämtad 2012-05-09.

[Fouc] _____, *Python library: Built-in types*, `http://docs.python.org/py3k/library/stdtypes.html`, Hämtad 2012-05-09.

[Foud] _____, *Python reference: Datamodel*, `http://docs.python.org/py3k/reference/datamodel.html`, Hämtad 2012-05-11.

[Foue] _____, *Python tutorial: Namespaces*, `http://docs.python.org/py3k/tutorial/classes.html#python-scopes-and-namespaces`, Hämtad 2012-04-20.

[Guia] GNU Guile, *Goops info manual: Class definition protocol.*, `https://www.gnu.org/software/guile/manual/html_node/Class-Definition-Protocol.html`, Hämtad 2012-05-13.

[Guib] _____, *Goops info manual: Simple conditional evaluation*, `https://www.gnu.org/software/guile/manual/html_node/Conditionals.html`, Hämtad 2012-05-14.

[Knu74] Donald E. Knuth, *Structured programming with go to statements*, ACM Comput. Surv. **6** (1974), 261–301.

[KR91] Gregor Kiczales and Jim Des Rivieres, *The art of the metaobject protocol*, MIT Press, Cambridge, MA, USA, 1991.

[Mog] Eben Moglen, *Why the fsf gets copyright assignments from contributors*, `https://www.gnu.org/licenses/why-assign.html`, 2010-04-26. Hämtad 2012-04-26.

[Sof] Black Duck Software, *Open source license data*, `http://osrc.blackducksoftware.com/data/licenses/index.php`, 2010-04-26. Hämtad 2012-04-26.

[Staa] Richard Stallman, *Copyleft: Pragmatic idealism*, `https://www.gnu.org/philosophy/pragmatic.html`, 2010-04-26. Hämtad 2012-04-26.

[Stab] _____, *What is copyleft?*, `https://www.gnu.org/philosophy/copyleft.html`, 2010-04-26. Hämtad 2012-04-26.

[Sta02] Richard M. Stallman, *Free software: Free society*, GNU Press, Boston (MA), USA, 2002.

[Wik] The Python Wiki, *Should i use python 2 or python 3 for my development activity?*, `http://wiki.python.org/moin/Python2orPython3`, Hämtad 2012-04-26.

[Winb] ———, *wingolog: ecmascript for guile*, `http://wingolog.org/archives/2009/02/22/ecmascript-for-guile`, 2009-02-22. Hämtad 2012-03-18.

Övrig litteratur

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, fourth edition, 2009.

[4] Ira R. Forman. Putting metaclasses to work. In *TOOLS 1998: 26th International Conference on Technology of Object-Oriented Languages and Systems, 3-7 August 1998, Santa Barbara, CA, USA*, page 447. IEEE Computer Society, 1998.

[5] Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006.

[6] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[7] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011.

[8] Andy Wingo. emacs-devel: guile and emacs and elisp, oh my! `https://lists.gnu.org/archive/html/emacs-devel/2010-04/msg00665.html`. 2010-04-14. Hämtad 2012-03-18.