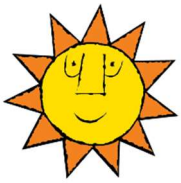


ÄLVKULLEGYMNASIET

Metoder för exekvering av programkod och dess effektivitet

Methods for executing program code and their efficiency

Gymnasiearbete vt-2023
Ludvig Magnusson, TE20c
Ansvarig lärare: Johansson Wåhlin, Astrid

Summary

The purpose of this project is to investigate how different representations of program source code, execution methods, compare against each other in terms of runtime performance. The methods tested are interpreting the program with an AST Interpreter, compiling it to bytecode and running it in a virtual machine, and finally compiling to x86 assembly code.

To do this, a programming language is built that has implemented the different execution methods that is used for the comparison. Six programs performing various math calculations are then ran and their runtime compared to each other. The primary source is a book on compilers authored by computer scientist Alfred V.Aho, among other various sources in form of writeups by people with experience in creating programming languages.

The research is limited to the mentioned methods, so for example Just-In-Time compilation is not investigated. Neither any optimizations unique to a specific execution method is implemented.

The study concludes that compiling the source code to x86 assembly code that is then assembled and ran has, by a large margin, the lowest runtime among the execution methods. The AST Interpreter outperformed the Bytecode Interpreter by a small margin in half of the tests and the other half had very similar performance, but in a few cases the Bytecode interpreter had marginally lower runtime than that of the AST Interpreter. The explanation for the similar performance of the Bytecode and AST Interpreter is not clear but could be explained by inefficiencies in the implementations. Potential future studies could more thoroughly investigate those two methods and ways of optimizing them further to achieve better performance.

Innehållsförteckning

1 Inledning

Programmering är en stor del av det idag tekniska i samhället. Det finns många programmeringsspråk som används idag med olika egenskaper, som att de är enkla att använda, enkla att förstå och att de exekverar kod snabbt. Ofta är det viktigt att körningstiden för koden är så låg som möjligt, vilket utöver mänskliga faktorer, är baserat på vilken metod som används för att exekvera den. Källkod är endast koden människor skriver, men för att en dator ska förstå och kunna göra något med den krävs en annan representation av den. Det finns många olika representationer, men de vanligaste är att kompilera till maskinkod som en processor sedan exekverar eller att kompilera till instruktioner som sedan körs i en virtuell maskin. Arbetet handlar om att skapa ett eget programmeringsspråk och undersöka hur olika representationer av källkoden, eller exekveringsmetoder, påverkar körningstiden för programmet.

1.1 Syfte och frågeställning

Syftet med arbetet är att undersöka hur olika tillvägagångsätt för exekvering av kod jämför sig i förhållande till varandra i avseende på körningstiden.
För att undersöka detta ska ett grundläggande programmeringsspråk skapas som en grund som jämförelserna baserar sig på.
Frågan är: *Hur skiljer sig exekveringstiden mellan körning av bytecode, kompilerad maskinkod och en direkt tolkning av syntaxträdet som representerar programmet?*

1.2 Metod

Arbetet har utgått ifrån litteratur i form av böcker och artiklar skrivna av personer med erfarenhet, för att för att skapa ett programmeringsspråk grundat i principer som är standard för programmeringsspråk idag. Det finns ingen tydlig primärkälla, men boken *Compilers: Principles, Techniques, and Tools* (2006) som är skriven av datavetaren Alfred V.Aho är grunden till det mesta detta arbete beskriver. Programmeringsspråket som skapats har möjligheten att köras genom de tre olika exekveringsmetoderna arbetet har undersökt. Prestandan mellan metoderna jämföras då för att komma till en slutsats. Det är viktigt att alla tre exekveringssätt kan köra ett givet program och ge samma resultat för alla, annars blir det utmanade att jämföra prestandan.

Körningstiden beror självklart på vilken dator programmen körs på, men eftersom detta arbete endast är en jämförelse mellan exekveringssätt så har det troligen inte någon påverkan på resultatet. Det finns dock en chans att en snabbare eller långsammare processor kan ge en av exekveringsmetoderna en fördel som den annars inte skulle haft. Detta kan möjligen ge felaktiga resultat och är inget som detta arbete har tagit hänsyn till. Med störst sannolikhet är det nog inte ett problem då ingen av exekveringssätten är optimerade för en viss processor eller dess egenskaper, men det går inte att veta säkert. Datorn programmen testades på har en Intel Core i5-4440 @ 3.10GHz.

Prestandan hos exekveringsmetoderna beror mycket på implementationen för tolkaren och kompilatorn i språkbehandlaren. Det är inte säkert att en annan implementation för samma körningsmetoder ger liknande svar och det är något som är väldigt svårt att undersöka generellt. Resultatet av detta arbete kan fortfarande användas för att komma till slutsatser om de olika exekveringssätten, eftersom det skapade programmeringsspråket och

språkbehandlaren baserar sig på redan kända principer ifrån personer som designat och implementerat samma sak tidigare.

För att jämföra körningstiden skrevs sex olika program som utför matematiska beräkningar och implementerar några algoritmer. Programmet kördes enligt de olika exekveringsmetoderna fem olika gånger var, där varje körning utgick ifrån en ”kall start” för att undvika att data sparas i cache-minnet. En kall start innebär att programmet stängs ner och startas på nytt inför varje försök. Alla program kompilerades också i *release mode* för att ge optimal prestanda.

Koden för att tolka det skapade språk skrevs i programmeringsspråket C++ och jag använde mig av utvecklingsmiljön Visual Studio 2019. Genererad assemblykod assemblerades av Netwide Assembler (NASM) och Simple ASM (SASM) användes för att felsöka assemblykod under utvecklingen. C++ är ett populärt språk som är någorlunda enkelt att skriva i men samtidigt ger mycket hög prestanda eftersom det är väldigt nära hårdvaran. En tolkare eller bytecode-maskin skriven i C++ ger en rättvisare jämförelse med körning av assemblykod just för att C++ är närmare hårdvaran. Om tolkaren till programmeringsspråket hade varit skriven i Java eller Python hade samma kod troligen gett längre körtid. För att ge den bästa möjligheten för att rättvist jämföra exekveringsmetoderna, så användes det programmeringsspråk som ger bättre prestanda.

1.3 Avgränsning

Effektiviteten för andra metoder för exekvering av kod, såsom JIT kompilering, kommer inte att undersökas i detta arbete. Även en kompilator som optimerar koden baserat på minnesanvändning och prestanda innan den körs kommer inte heller att undersökas. Många kompilatorer idag använder avancerade metoder för att optimera kod bättre än en människa skulle kunna skriva den, men skulle detta implementeras i mitt språk skulle det snarare bli en undersökning om hur de påverkar prestandan. Många optimeringar är exklusiva till vissa exekveringsmetoder, därav ger det en felaktig jämförelse av prestandan mellan dem. Slutligen, med termen prestanda menas tiden det tar för programmet att köras från start till slut och tiden börjar klockas när koden för programmet börjar exekveras.

1.4 Ordförklaring

AST – Förkortning för abstrakt syntaxträd, samma sak som ett syntaxträd.

Syntaxträd - En representation av källkoden för ett program i form av ett träd.

Polett – Ett ”ord” som en lexer har tolkat.

Interpreter – Tolkar något, exempelvis instruktioner, för att ge ett resultat.

Kompilator – Gör om källkod till instruktioner, ofta till instruktioner i assemblyspråket.

Assemblerare – Gör om assemblyinstruktioner till maskinkod.

Bytecode instruktion – En instruktion som kan köras av en bytecode interpreter.

Assembly – Ett människovänligt språk skrivet med instruktioner för en viss processorarkitektur.

Opcode – Namnet på en bytecode eller assemblyinstruktion som avgör vad instruktionen gör.

Maskinkod – Assemblyinstruktioner som gjorts om till numeriska värden som sedan processorn kan exekvera.

Stack – En hög som värden kan läggas till och tas bort ifrån.

LIFO kö – En ködatastruktur där det senaste värdet som lades till är det som tas bort.

Register – Ett litet minnesområde i processorn där ett litet värde kan sparas för att användas av instruktioner.

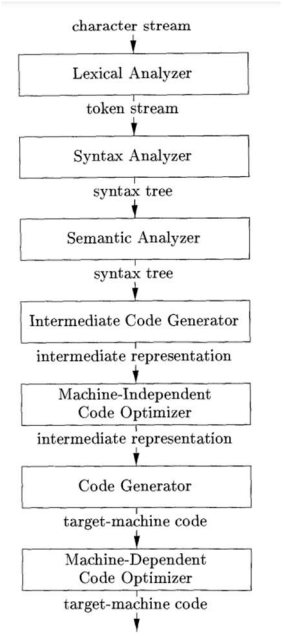
Pusha – Ett värde läggs högst upp i stacken.

Poppa – Värdet högst upp i stacken tas bort.

Release mode – C eller C++ kompilatorn använder optimeringar för att generera ett mer optimerat och snabbare program.

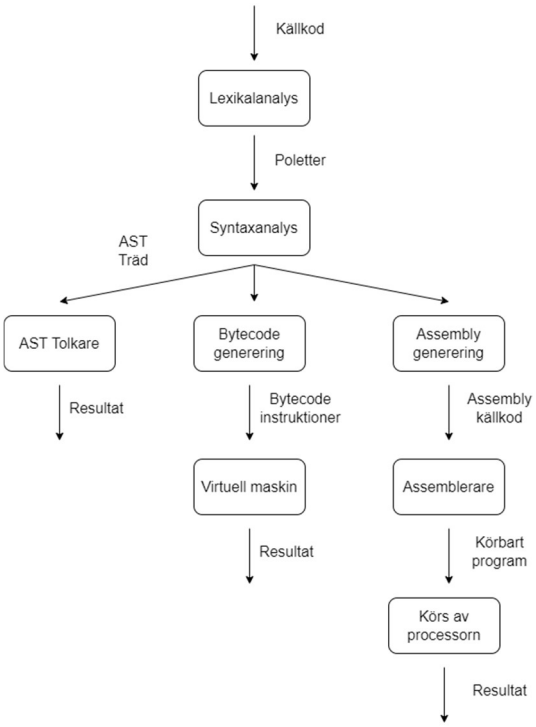
2 Strukturen av en Språkbehandlare

För att gå från källkod till något som en dator kan exekvera så krävs det ett program som tar in källkoden. Programmet kallas generellt för en *språkbehandlare* (*language processor* på engelska) och fungerar genom att översätta källkoden till kod i en annan form som kan exekveras av en dator. Ifall det språkbehandlaren skapat är instruktioner för någon typ av maskin, så kallas språkbehandlaren för en *kompilator* och koden har *kompilerats*. Ifall programmet tolkas utan att ha gjorts om till maskininstruktioner, så kallas det för en *interpreter*. Oavsett är den internt uppbyggd i faser, där varje fas tar in data och ger ut ett resultat, som sedan matas in i nästa fas (Alfred V. Aho, 2006). Se *figur 0*.



Figur 0. Överblick över ett exempel för strukturen av en kompilator (Alfred V. Aho, 2006, s 5).

Alla typiska kompilatorer och tolkare har två faser som alltid är gemensamma; lexikalanalys och syntaxanalys. Utöver så finns den viktiga kodgenereringsfasen som sker efter syntaxanalysen, men endast en kompilator har den. Se *figur 1* för hur faserna ser ut för en enkel språkbehandlare som har en AST tolkare, kör bytecode och genererar kompilerad maskinkod.



Figur 1. Överblick över hur en språkbehandlare för de tre olika exekveringssätten är strukturerad.

2.1.1 Lexikalanalys

Det första steget för en språkbehandlare är att plocka ur beståndsdelarna i källkoden, ungefär som att hitta fraser, ord och adjektiv i en mening. Detta görs av en så kallad l*exer* som analyserar och känner igen orddelar i källkoden (Nationalencyklopedin, u.d.). Den läser av alla tecken i källkoden enskilt, men med sammanhang kan den hitta fraser och vad de har för syfte i programmet.

Exempelvis, ur denna mening, *"engrönödlahopparlångt",* så skulle en lexer kunna ge varje ord för sig självt, vilket underlättar med att läsa och förstå meningen. Det är liknande i en språkbehandlare som till exempel ska kompilera källkoden $int\ v = 1 + 2 * 3$. Lexern plockar ut de olika beståndsdelarna och gör det tydligt vad varje del av källkoden ska representera, som att int syftar på en datatyp och v är namnet på en variabel. Från datorns synvinkel är detta bara enskilda tecken om inte lexern hade funnits och gett en tydlig mening till allt. Ett "ord" som en lexer tolkat kallas för en *polett (*på engelska, *token)* och den innehåller information om vilken typ av polett det är, exempelvis variabel eller datatyp, och även vilket värde den har, som ett siffervärde eller ett namn (se *Tabell 1*). Förslagsvis sparar en token av typen *Number* också siffervärdet för numret.

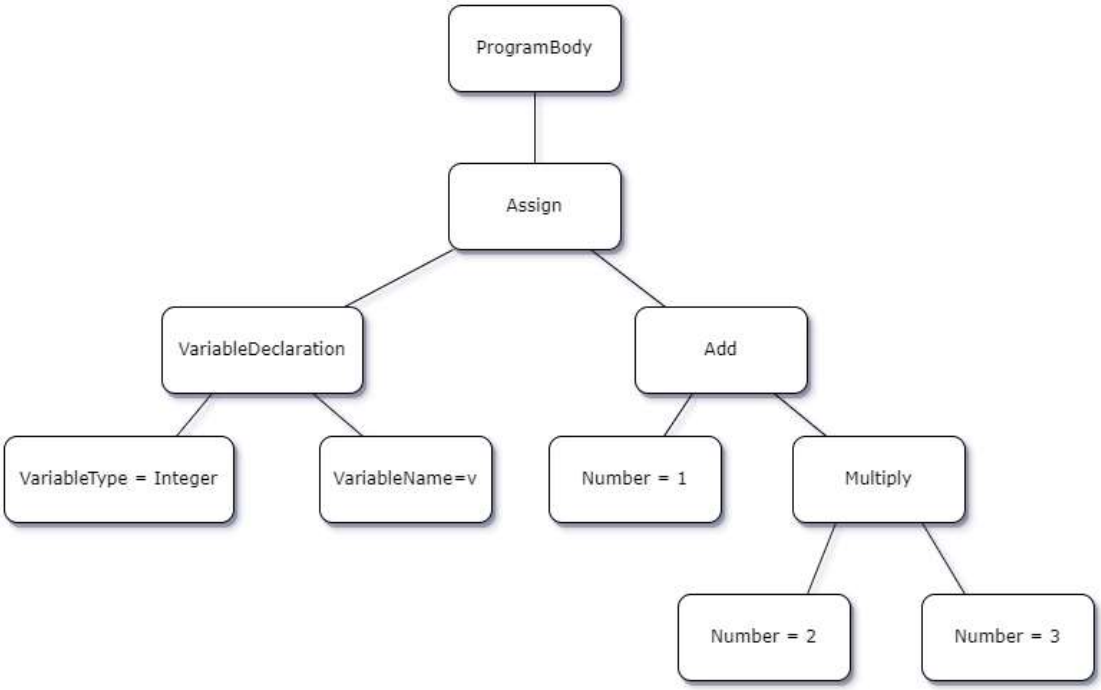
Tabell 1. Tabell med exempel på poletter för källkoden $int\ v = 1 + 2 * 3$

Typ	Värde
VariableType	int
Variable	v
Assign	=
Number	1
Add	+
Number	2
Multiply	*
Number	3

2.1.2 Syntaxanalys

Nästa steg i processen är att skapa en representation av hela programmet utifrån poletterna lexern genererat. Lexikalanalyser ger alla orddelar av källkoden, men det är syntaxanalysens uppgift att sätta ihop dem till något som berättar hur programmet ska tolkas, vilka poletter som tillhör vad och ser till att olika saker sker i rätt ordning i programmet, exempelvis addition och multiplikation.

För att göra det så används ett abstrakt syntaxträd, (även kallat syntaxträd), som representerar den hierarkiska språkstrukturen av källkoden till programmet och det är en träddatastruktur (Alfred V. Aho, 2006). Syntaxträdet kan jämföras med ett stort träd i verkligheten. Stammen till trädet är grunden alla grenar sitter fast i. De tjocka grenarna sittandes i stammen har den som sin förälder, vilket också innebär att grenarna är "barn" till stammen. De tjocka grenarna har flera mindre grenar anslutna till sig själv, som i sin tur kan ha fler grenar och så vidare tills det bara är löv kvar på grenen. Skillnaden mellan ett syntaxträd och ett verkligt träd är att i datatermer så kallas stammen till trädet för *roten* och grenarna för *noder*. Ett syntaxträd börjar med en rot som har ett antal noder som barn men ingen förälder (se *figur 3*). En nod har noll, en eller flera andra noder som barn, som i sin tur kan ha fler noder och så vidare tills det inte finns några fler. En nod innehåller information om vilken typ av nod det är men även vilket värde den har.



*Figur 3. Ett exempel på ett syntaxträd till källkoden "int v = 1 + 2 * 3"*

I *figur* 3 så tolkas trädet som att det som är längst ner i trädet är det som händer först under exekveringen av koden. Det kan ses på högra sidan av lika-med tecknet i trädet. Resultaten av noden till vänster om additionsnoden adderas med resultatet av det högra, men för att kunna göra det så måste först det till höger vara känt. Detta medför att multiplikationen utförs först, additionen sist och resultatet sparas slutligen i variabeln till vänster, vilket är precis det vi vill uppnå med trädet; att göra om källkoden till ett format som förmedlar strukturen av programmet.

2.1.3 Kodgenerering

Med syntaxträdet till hands kan en dator förstå programmet mycket bättre jämfört med ifall endast källkoden var tillgänglig. Trädet innehåller tillräckligt med information om strukturen av programmet så att det skulle kunna tolkas direkt av ett program genom att bara följa grenarna. Ett alternativ till att detta är att en kompilator gör om varje av trädets noder till instruktioner som i stället kan tolkas av *processorn*, till skillnad från att ett *program* tolkar noderna. Detta är en viktig fas av en språkbehandlare. Ett syntaxträd kan tolkas av ett program skrivet för att göra det, men det kan inte exekveras som instruktioner för processorn i datorn. Därför behövs kodgenereringen som skapar instruktioner processorn faktiskt förstår, i stället för en generell representation av strukturen för källkoden. Mer om detta beskrivs i Kapitel 3.2 och 3.3.

3 Exekveringsmetoder

En exekveringsmetod är den delen av en språkbehandlare som faktiskt kör källkoden, för att exempelvis utföra matteoperationer, logik eller skriva ut en sträng till konsolen. I detta kapitel kommer metoderna AST interpreter, Bytecode interpreter och kompilering till maskinkod förklaras hur de fungerar.

3.1 AST Interpreter

Som ordet *interpreter* antyder, så innebär en AST Interpreter att det skapade syntaxträdet tolkas av ett datorprogram kallat för en *interpreter*, eller *tolkare* på svenska. Varje nod tolkas och fungerar som en instruktion för tolkaren och talar om vad som ska hända. Koden för en tolkare fungerar genom att en funktion tar in en nod till syntaxträdet som ett argument och ger alltid tillbaka ett värde. För varje typ av nod så körs olika koder för att göra det som noden kräver. Om noden exempelvis är av typen *addition*, så ska de två barn-noderna adderas och resultatet returneras.

Fördelen med en AST tolkare är att de är väldigt simpla att implementera. Tolkaren startar högst upp i syntaxträdet och tolkar varje nod genom att använda rekursion. AST tolkare är inte så vanliga som exekveringsmetod för programmeringsspråk då andra körningssätt ibland erbjuder mer möjligheter att optimera prestandan, men AST tolkare behöver inte vara långsamma eller sakna möjlighet för optimering (Stefan Marr, 2014). Operativsystemet *Serenity OS* har en egen implementation av en JavaScript tolkare vid namn *LibJS*, som använder sig av en AST tolkare för att köra JavaScript-kod. (Kling, 2023)

3.2 Bytecode Interpreter

Ett alternativ till en AST interpreter är att linjärisera syntaxträdet till att representera programmet som en följd av instruktioner (Stefan Marr, 2014). Dessa instruktioner kallas för *bytecode instruktioner* och tolkas av en virtuell maskin skapad för att förstå dessa instruktioner. Instruktionerna har många likheter med assemblyspråkets instruktioner, men i stället för att använda instruktioner för en specifik processor, så är instruktionerna specialgjorda för en abstrakt processor som vi är fria att definiera hur vi vill (se *figur 4*). Fördelen med att använda en abstrakt processor är att oavsett vilken dator koden körs på, så kommer bytecode alltid kunna köras eftersom den är universell.

```
1 push 2
2 push 3
3 add
4 jump 1
```

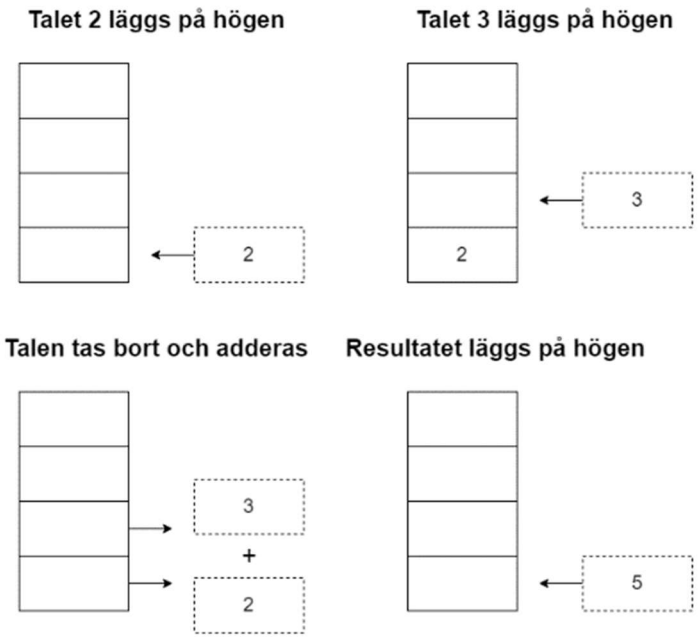
Figur 4. Exempel på ett program skrivet med bytekodinstruktioner. Programmet adderar 2 och 3 tills att programmet stängs ner

Den virtuella maskinen börjar högst upp i *figur 4* och läser av varje instruktion i ordningen de är i och går vidare till nästa instruktion när den nuvarande instruktionen har körts klart. Maskinen kan också hoppa fritt till en annan instruktion i programmet som kommer senare eller som redan körts. Det är användbart för att kunna konstruera if, while och for-loopar, men också för att anropa funktioner.

3.2.1 Stackbaserad virtuell maskin

Det finns olika typer av virtuella maskiner för att tolka instruktionerna. Den enklaste är en så kallad stackbaserad maskin och programmeringsspråket Java använder bland annat en sådan för sin exekvering (Anouti, 2018). Principen bygger på att det finns en ”hög” som det kan läggas till och tas bort värden ifrån. När ett värde läggs till så hamnar det högst upp i högen, och när ett värde ska tas bort så försvinner alltid det som senast lades till. Denna typ av datastruktur kallas för en *LIFO kö*, eller, *sist in, först ut kö* (*last in, first out queue* på engelska) (Wikipedia, 2022).

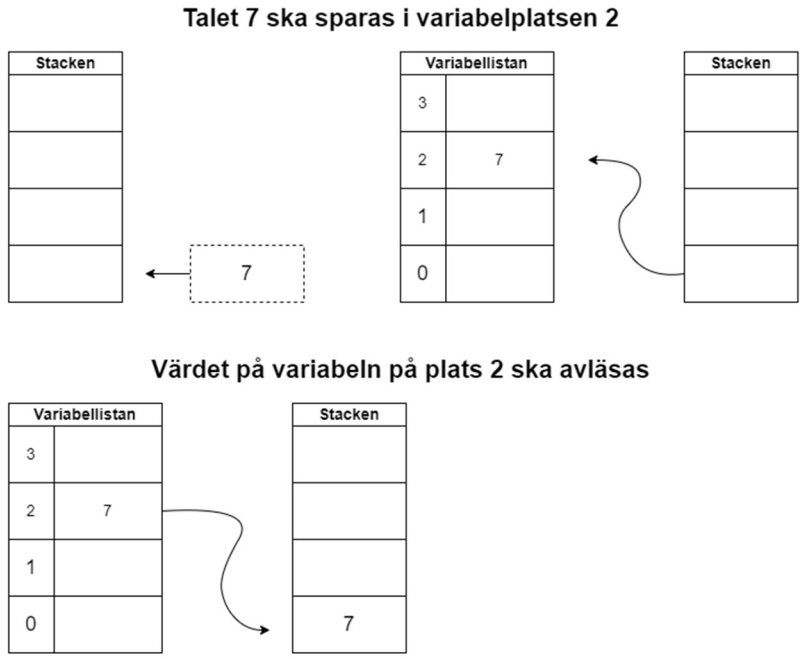
Stacken, eller högen, har sitt syfte att skicka och spara information mellan instruktioner under körning av programmet. I exempelprogrammet i *figur 4* så läggs talen 2 och 3 till i stacken. När maskinen ska tolka instruktionen *add* så vet den vad som ska adderas, för den tar ner de två värdena högst upp på stacken. Stacken blir då tom, eftersom värdena försvinner när de blir nedplockade för att användas. Slutligen läggs resultatet för additionen upp på stacken, i detta fall talet 5, för att senare kunna användas av andra instruktioner. *Se figur 5*



Figur 5. Bild över hur stacken ser ut när talen 2 och 3 ska adderas

3.2.2 Variabler i en virtuell maskin

En metod för att implementera variabler för en virtuell maskin är att ha varje variabelnamn motsvara ett index i en tabell. Med det indexet kan värden sparas till minnet på ett ställe och även avläsas därifrån. Se *figur 6* för hur det skulle kunna se ut.



Figur 6. Bild över när en variabel ska tilldelas ett värde och när värdet på en variabel ska avläsas

3.3 Maskinkod

Till skillnad från de andra exekveringssätten, så är *maskinkod* den enda som inte behöver ett separat program som tolkar den för att få saker att hända. Varje instruktion får processorn i datorn att göra en väldigt specifik sak, som att manipulera processorns register, värden i ramminnet eller att utföra aritmetik (Wikipedia, 2022). I stället för att använda en tolkare som tolkar någon representation av programmet, så kompileras i stället syntaxträdet till instruktioner i det människoläsbara assemblyspråket. De instruktionerna *assembleras* sedan till maskinkod bestående av endast numeriska tal, som inte är människoläsbara, tillskillnad från assemblyspråket. Dessa kan sedan processorn exekvera (Wikipedia, 2022)

Assemblykod liknar ett program skrivet med bytecode instruktioner. Instruktionerna är korta och utför ofta en liten uppgift. Exekveringen följer ordningen instruktionerna kommer i, men kan också hoppa runt fritt i programmet för att skapa loopar. Assemblykod kan använda processorns *stack* som fungerar i princip identiskt till den som en stackbaserad virtuell maskin använder. De största skillnaderna är att det även finns *register* på processorn som går att använda för att skicka runt data, men även att assemblykod inte kan exekveras på alla olika processorer. Varje processorarkitektur har olika uppsättningar instruktioner och är inte kompatibla med varandra. Se *figur 7* för hur instruktioner och användning av processorns register kan se ut. Själva kodgenereringen för att översätta syntaxträdet till assemblyinstruktioner är i stora drag samma process som den för att skapa bytecode instruktioner för programmet. Det som skiljer sig åt är vilka instruktioner som skapas, men processen är den samma.

```
1 section .text
2 global CMAIN
3 CMAIN:
4   mov eax, 400
5   mov ebx, 20
6
7   add eax, ebx
8
9   ret
```

Figur 7. Bild över enkelt program skrivet i x86 assembly som adderar talen 400 och 20 och sparar resultatet i eax registret.

4 Ö++

Språket som skapats för detta arbete kallas för Ö++ och är ett statiskt och starkt-typat språk med stöd av grundläggande datatyper, logik och stöd för att definiera egna funktioner. Språket har som fördel att oavsett vilken källkod som skrivs så kan den tolkas av en AST interpreter, göras om till bytecode och köras i en virtuell maskin eller kompileras till x86 maskinkod. Syntaxen liknar den i C-språket och känns bekant för någon med erfarenhet av programmering. Se exempelprogrammet i *figur 8* för en överblick över syntaxen.

```
// Slumpat flyttal mellan min och max
float rand_range_float(float min, float max) => {
    return (max - min) * (to_float(rand()) / 32767.0) + min;
};

// Estimera Pi med Monte Carlo alorithmen
float pi() => {
  int inside = 0;
  int iterations = 1000;

  for (int i = 0, i < iterations, i++) {
    float x = rand_range_float(-1.0, 1.0);
    float y = rand_range_float(-1.0, 1.0);

    if (x * x + y * y <= 1.0) {
      inside++;
    };
  }

  return 4.0 * to_float(inside) / to_float(iterations);
};

srand(time());

print("pi = %f\n", pi());
```

Figur 8. Bild över ett program skrivet i Ö++ som estimerar talet pi.

4.1.1 Vad kan språket göra?

Språket har stöd för variabler av typen heltal, flyttal (double precision) och strängkonstanter. Variabler är inte konstanta men är kan bara tilldelas värden som har samma typ. I *figur 9* så visas exempelkod för detta.

```
int a = 9
float b = 13.456
string c = "Hello world!"
```

Figur 9. Variabler av olika datatyper

Variabler kan vara antingen lokala eller globala. Lokala variabler skapas när exekveringen kommer till de och finns bara tillgängliga inom den nuvarande måsvingen. Globala variabler skapas det första som sker under körningen och finns alltid tillgängliga. Se *figur 10* för exempelkod.

```
global int pi = 2

{
  string local = "Hello";
}
```

Figur 10. Globala och lokala variabler

Variabler kan även ökas och minskas med en i taget. Detta fungerar endast för heltal och flyttalsvariabler, inte för strängar. Se *figur 11*.

```
1 int i = 0;
2 i++;
3 i += 4;
```

Figur 11. Ökning av en heltalsvariabel

Språket kan också utföra aritmetik, som addition, subtraktion, multiplikation och division. Matematiska uttryck kan tolkas och prioriteringsreglerna gäller. Se *figur 12*.

```
1 int v = 51 / ((1 + 3) * 5 - 3);
```

Figur 12. Aritmetik och tolkning av ett matematiskt uttryck

De grundläggande for och while-looparna som förväntas finns och även if-satser. For-loopen skiljer sig i att i stället för att semikolon separerar de tre delarna, så gör kommatecken det. De grundläggande jämförelseoperatorerna mellan tal finns och fungerar i samband med en if-sats. Se *figur 13*. I samband med en if-sats så stöds också *else* för att exekvera kod om villkoret inte uppfylldes. Se *figur 14*.

```
1 float i = 0;
2 while (i <= 10.0) {
3  if (i >= 3.9) {
4     print("i >= 3.9");
5  };
6  i += 1.0;
7 }
```

Figur 13. If, while, for satser och jämförelse mellan tal

```
1 if (1 == 2) {
2   print("false");
3 } else {
4   print("true");
5 };
```

Figur 14. Else i samband med en if-sats

Språket stödjer även funktioner som liknar de i C-språket. En funktion har ett frivilligt returvärde och kan ta in ett antal argument. Ifall funktionen har ett returvärde så måste ett värde returneras, men är returvärdet *void* så behövs det inte. I språket finns ett antal inbyggda funktioner som kan anropas, men det går även att definiera egna. Vart i programmet funktionen definieras har ingen betydelse, (så länge den inte är inom en måsvinge), för den kan anropas från vart som helst. Rekursion stöds dessutom, vilket gör det möjligt att exempelvis skriva ett program som beräknar fakulteten av ett tal, se *figur 15*.

```
1 int factorial(int n) => {
2   if (n <= 1) {
3     return 1;
4   }
5   return n * fact(n - 1);
6 };
```

Figur 15. Funktionen factorial beräknar fakulteten av talet n

4.1.2 Lista över inbyggda funktioner

Ö++ har ett antal inbyggda funktioner som kan anropas. Dessa kan vara funktioner som redan finns i C-standard biblioteket, men även andra funktioner som gör något som inte finns inbyggt. I detta underkapitel står funktionerna som finns i språket. Detta är för att visa på vad språket kan göra, men också som en dokumentation för att kunna förstå de bifogade exempelprogrammen.

4.1.2.1 print

Funktionsprototyp: $void\ print(string, \ldots args)$
Förklaring: Från C standardbiblioteket. Skriver ut argumenten till terminalen utan en ny rad.

4.1.2.2 rand

Funktionsprototyp: $void\ rand()$
Förklaring: Från C standardbiblioteket. Genererar ett slumpmässigt heltal mellan 0 och 0x7fff.

4.1.2.3 srand

Funktionsprototyp: $void\ srand(int\ seed)$
Förklaring: Från C standardbiblioteket. Sätter ett specifikt frö till *rand()* funktionen

4.1.2.4 time

Funktionsprototyp:$int\ time()$
Förklaring: Från C standardbiblioteket. Returnerar antalet sekunder sedan första januari 1970.

4.1.2.5 sin, cos, tan

Funktionsprototyp: $float\ sin(float\ angle)$
Förklaring: Sin, cos och tan fungerar på samma sätt, men returnerar värdet för sin respektive funktion. Vinkeln *angle* är i radianer.

4.1.2.6 sqrt

Funktionsprototyp: $float\ sqrt(float\ number)$
Förklaring: Beräknar kvadraten för talet *number*

4.1.2.7 pow

Funktionsprototyp: $float\ pow(float\ base, float\ exponent)$
Förklaring: Beräknar talet *base* upphöjt till *exponent*

4.1.2.8 abs

Funktionsprototyp: $float\ abs(float\ number)$
Förklaring: Beräknar absolutbeloppet av talet *number*

4.1.2.9 to_int

Funktionsprototyp: $int\ to\_int(float\ number)$
Förklaring: Konverterar flyttalet *number* till ett heltal

4.1.2.10 to_float

Funktionsprototyp: $float\ to\_float(int\ number)$
Förklaring: Konverterar heltalet *number* till ett flyttal

4.2 Skillnader

Ö++ är skapat i stora drag utifrån den generella implementationen av de tre olika exekveringsmetoderna. Jag har dock tagit mig friheten att försöka lösa många av problemen som uppstått själv i stället för att söka hur andra implementationer gjort. Det har lett till att implementationerna skiljer sig ifrån den generella strukturen i vissa fall. I detta delkapitel kommer min implementation för de olika exekveringsmetoderna redovisas övergripligt.

4.2.1 AST Interpreter

Tolkaren tolkar syntaxträdet direkt efter att det skapats, utan att göra djupare analyser av koden. Exempelvis, om en egendefinierad funktion anropas så går tolkaren igenom noderna som om den aldrig sett den förut. En del onödiga data tolkas alltså flera gånger, som exempelvis typverifieringen som sker när ett värde ska tilldelas till en variabel.

För varje måsvinge i källkoden skapar tolkaren en så kallad *scope frame*. Alla *scope frames* finns sparade i tolkaren i en *LIFO kö* där frames *pushas* (läggs till högst upp i högen) och *poppas* (det översta i högen plockas ner). En scope frame innehåller alla variabler det nuvarande programstycket ska ha tillgång till. Variablerna sparas i en *hashmap* där nyckeln är namnet på variabeln och värdet är värdet på själva variabeln. Se *figur 16*. Variabler från tidigare scopes kopieras till den nuvarande så att de fortfarande kan vara tillgängliga. När scopet har tagit slut så kopieras de variabler som har förändrats till det scopet som var innan, så att ändringar av variabler som inte är lokala till scopet går förlorade. Sedan så poppas scopet och försvinner för att inte användas igen.

Ett värde kan vara av flera olika datatyper, såsom heltal, flyttal eller en sträng.
En klass som kallas *Value* används så att ett värde kan användas till funktioner utan att den underliggande datatypen på värdet spelar roll. Klassen har en variabel var för de olika möjliga underliggande datatyperna, men även vilken av datatyperna instansen faktiskt använder, så att rätt variabel kan avläsas när värdet behövs. Utöver det så sparas också en flagga som indikerar om variabeln är en global variabel eller inte, se *figur 16*

```
1 class Value {
2   int intData
3   float floatData
4   string stringData
5
6   bool isGlobal;
7   Enum variableType;
8 }
9
10 class ScopeFrame {
11   hashmap<string, int> variables
12 }
```

Figur 16. Psuedokod för klassen Value och ScopeFrame som används i AST tolkaren

4.2.2 Bytecode Interpreter

Efter att syntaxträdet skapats så kompileras det till instruktioner för den virtuella maskinen. En instruktion är uppbyggd av en *opcode* och ofta ett antal *argument*. *Opcoden* är den viktiga delen som berättar för maskinen vad som ska ske. En opcode är ofta kort och är en förkortning för vad den innebär. De som finns i Ö++ är inspirerade av opcodes som JVM

(Java Virtual Machine) implementerar. Argumenten används för att ge data som instruktionen behöver, exempelvis om ett visst värde ska läggas på högen så är det ett argument, se *figur 17*.

Opcode	Argumenttyp	Förklaring
push_number	Number	Pushar argumentet på stacken
store	Number	Poppar värdet på stacken och sparar det i indexet givet av argumentet
load	Number	Pushar värdet på stacken som är sparat i indexet givet av argumentet
add		Poppar de två talen högst upp på stacken, adderar de och pushar resultatet på stacken
cmpgt		Poppar de två talen högst upp på stacken. Om tal 1 är större eller lika med tal 2 läggs en *etta* på stacken, annars en *nolla*.
jmp_if_true	Number	Poppar talet på stacken och ifall det är *ett* så hoppar exekveringen till instruktionen på platsen angivet av argumentet

Figur 17. Tabell över ett urval av instruktioner från bytecode tolkaren med förklaring

Likt AST tolkaren, så för varje måsvinge pushas en motsvarighet till *scope frame* på *LIFO kö*. Den fungerar identiskt som den i AST tolkaren, men varje *frame* innehåller också en stack. Det är högen som en stackbaserad virtuell maskin använder för att skicka runt data, som exempelvis Java gör. Variabellistan och stacken har en förbestämd maximal storlek, till skillnad från den enklare variabel hashmappen i AST tolkaren. Fördelen med en statisk storlek är att det går snabbare att lägga till och ta bort data, men ska en ny frame skapas eller en gammal kopieras så tar det längre tid. Detta eftersom storleken behöver vara ganska stor för att inte riskera att få slut på platser för data. Detta skulle möjligen kunna orsaka körningstiden negativt och är en sak med arkitekturen som eventuellt kan vara värd att ändra.

Stacken består av instanser av klassen *Value* som då är ett heltal eller flyttal. När jag bestämde arkitekturen för maskinen så tog jag mycket inspiration av hur Javas maskin är uppbyggd. Där är varje plats i högen endast 4 bytes lång. Alla andra datatyper, som strängar, fält eller objekt som inte är tal är sparade på ett *heap* (Oracle, 2015). Heapet är en del av den virtuella maskinens ramminne som kan allokeras och användas utan samma begränsningar som minnet i stacken innebär. I stället för att vara begränsad till data som är 4 eller 8 bytes stor, så kan data på heapet vara så stora de behöver. Eftersom högen är ganska begränsad i storlek så passar det väldigt bra för exempelvis strängar, som väldigt enkelt fyller upp högen ifall de har många karaktärer. Därför finns det i Ö++, utöver stacken och variabellistan, också en klass vid namn *Heap* som sparar större data (se *figur 18*). En sträng allokeras och dess pekare sparas i ett objekt av typen HeapEntry, som sedan sparas i klassen Heap. Det numeriska värdet på pekaren till strängen kan då sparas i variabler och användas för funktioner, och värdet kan vid tillfälle användas som en pekare till den underliggande strängen, om den exempelvis ska skrivas till konsolen.

```
1  struct HeapEntry
2  {
3      int m_Type = 0;
4      int m_Id = 0;
5
6      void* m_Data = nullptr;
7  };
8
9  struct Heap
10 {
11     std::unordered_map<int, HeapEntry> m_Entries;
12
13     int m_NextFreeId = 0;
14
15     HeapEntry& CreateString(const std::string& str);
16 };
```

Figur 18. Hur klasserna Heap och HeapEntry är uppbyggda (Heap.h)

```
1  HeapEntry& Heap::CreateString(const std::string& str)
2  {
3      int id = m_NextFreeId;
4      m_Entries[id] = HeapEntry(0, id, (const char*)nullptr);
5
6      char* strData = CopyString(str.c_str());
7
8      m_Entries[id].m_Data = (void*)strData;
9
10     // Increment to use a different slot for the next object
11     m_NextFreeId++;
12
13     return m_Entries[id];
14 }
```

Figur 19. Hur en sträng allokeras och sparas i den virtuella maskinens minne (Heap.cpp)

4.2.3 Assembly

Ö++ använder x86 assembly i stället för den mer moderna x64. Anledningen till det är att det finns mer resurser om x86 arkitekturen eftersom det har funnits mycket längre, men också att de källorna jag hittade som verkade bra behandlade endast x86 arkitekturen. Det mest intressanta att nämna om den genererade assemblykoden är hur flyttal hanteras. I AST och bytecode tolkarna hanteras flyttal på i princip samma sätt som ett heltal, men så fungerar det inte i assembly. Normala heltal kan läggas i processorns register, exempelvis *eax*, och sen manipuleras av en instruktion som *inc*. Flyttal kan inte det eftersom x86 arkitekturen kommer från en tid när det inte fanns inbyggt stöd för flyttal i processorn.

Den första dedikerade processorn som behandlade flyttals aritmetik var Intels 8087 co-processor. Alla moderna x86 processorer har än idag en inbyggd FPU (*floating point unit*) som kallas lite löst för x87. Den har en egen stack kallad för x87 högen och speciella instruktioner som endast arbetar med flyttal som finns i x87 högen (Wikibooks, 2022). Det utgör en liten utmaning med kodgenereringen. Kompilatorn behöver vara medveten om vilka typer av värden som ska hanteras då heltal och flyttal inte kan blandas. Detta är för att de är inkompatibla med varandra eftersom heltalen sparas i x86 stacken och flyttal i x87 stacken. Lyckligtvis är det inte det svåraste att lösa, men det krävs separata instruktioner och kodgenerering för de olika datatyperna överallt vilket orsakade att väldigt mycket tid än väntat lades ner på det.

5 Resultat

Frågan som detta arbete ska besvara är hur exekveringstiden mellan körning av bytecode, kompilerad maskinkod och en direkt tolkning av syntaxträdet som representerar programmet skiljer sig åt. Detta ska besvaras genom att skriva sex olika testprogram i Ö++ och jämföra körningstiden för de tre exekveringsmetoderna som jag implementerat. Körningstiden beror självklart på vilken dator programmen körs på, men eftersom alla program körs på samma dator så har det inte någon påverkan på resultatet. Processorn programmen testades på är en Intel Core i5-4440 @ 3.10GHz.

De sex olika programmen utför matematiska beräkningar och implementerar några algoritmer. Varje program kördes enligt de olika exekveringsmetoderna fem olika gånger var för att få en genomsnittlig tid, där varje körning utgick ifrån en ”kall start” för att undvika att data sparas i cache-minnet. En kall start innebär att programmet stängs ner och startas på nytt inför varje försök. AST tolkaren och Bytecode maskinen kompilerades också i *release mode* för att ge optimal prestanda för en rättvis jämförelse med den genererade assemblykoden. Källkoden för testprogrammen finns bifogade i Kapitel 7.

5.1 Körningstid

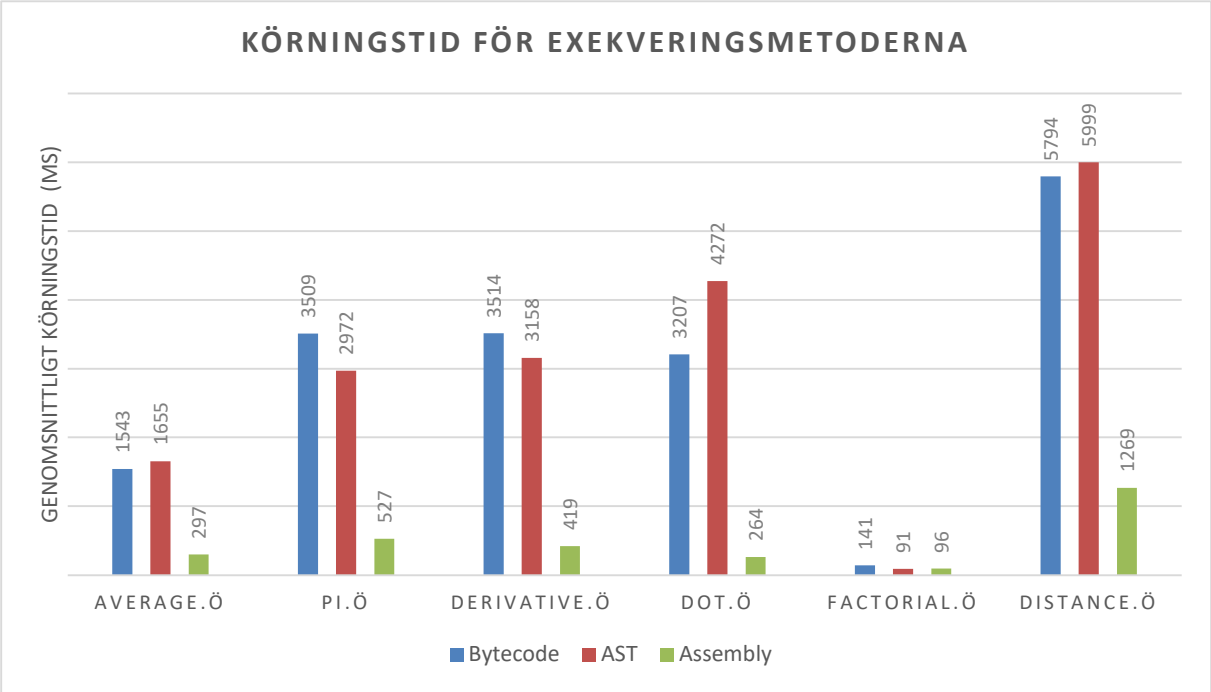


Diagram 1. Genomsnittlig körningstid för de olika programmen med olika exekveringsmetoder i Ö++. Lägre är bättre

Som kan ses i *diagram 1* presterade assemblykoden avsevärt mycket bättre än de andra två. AST och bytecode var ganska lika i prestanda i flera av testprogrammen. AST tolkaren presterade något bättre än bytecode tolkaren i hälften av testerna, men i *dot.ö* så presterade bytecode bättre med stor marginal. Bytecode presterade också bättre än AST tolkaren i *average.ö* och *distance.ö*, men med så liten marginal att de nog kan ses prestera lika.

6 Slutsatser och avslutande diskussion

Utifrån resultatet kan slutsatsen dras att kompilera källkoden till assembly ger avsevärt överlägsen prestanda. Jag var nästan helt säker på att det skulle bli så, men inte med sådan stor marginal. Anledning till att det är så är för att de andra exekveringsmetoderna har väldigt mycket mer ”overhead” eftersom en tolkare måste utföra mycket beräkningar och annat för att kunna tolka en instruktion. Att tolka en bytekodinstruktion motsvarar flera hundra genererade assemblyinstruktioner från C++ koden, men att kompilera Ö++ källkoden till assembly kan resultera i endast några få instruktioner för att utföra samma sak. Färre assemblyinstruktioner leder nästan alltid till snabbare körning.

Bytecode körningen var mycket långsammare än jag förväntade mig. Många virtuella maskiner med hög prestanda använder bytecode, exempelvis Java som Ö++ tagit inspiration ifrån, så det är lite förvånande att den presterade sämre än väntat. Jag skulle tro att min bytecode tolkare kan få bättre prestanda än AST tolkaren ifall instruktionerna skulle optimeras och minska hur mycket kod som behövs för att tolka en instruktion. Just nu sker mycket typverifiering under körningen, vilket lätt skulle kunna undvikas. De stora *Value* och *ScopeFrame* klasserna är också onödigt prestandakrävande när de ska kopieras och konstrueras på grund av dess storlek i minnet. Det som tolkaren gör mest är att pusha och poppa värden och frames, så ifall det skulle optimeras tror jag på en märkbar prestandaökning. Varför AST tolkaren presterade bättre än bytecode i *pi.ö* och *derivative*.ö, men inte *dot.ö* var oväntat. Jag har ingen bra förklaring för varför det skulle kunna komma sig. För att undersöka detta vidare skulle antingen profileringsmjukvara kunna användas, eller att skriva fler testprogram för att hitta vad som får körningssätten att skilja sig åt.

För att avsluta, att kompilera Ö++ källkod till assembly ger väsentligt bäst körningstid.
Att låta en AST tolkare tolka syntaxträdet ger liknande prestanda till att kompilera källkoden till bytecode instruktioner som körs av en virtuell maskin. Det finns en skillnad mellan de två, speciellt för vissa program och det skulle kunna undersökas vidare. Även hur det går att optimera dessa två metoder för att komma fram till var flaskhalsen för prestandan finns kan undersökas.

7 Referenser

Alfred V. Aho, M. S. (2006). *Compilers: Principles, Techniques, and Tools.* Pearson Education, Inc.

Anouti, M. (den 25 Mars 2018). *Introduction to Java Bytecode*. Hämtat från DZone: https://dzone.com/articles/introduction-to-java-bytecode

Evans, D. (2006). *x86 Assembly Guide*. Hämtat från University of Virginia Computer Science: https://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Kling, A. (den 14 03 2023). *Serenity OS*. Hämtat från https://github.com/SerenityOS/serenity

Lawlor, D. (u.d.). *The Stack: Push and Pop*. Hämtat från University of Alaska Fairbanks: https://www.cs.uaf.edu/2015/fall/cs301/lecture/09_16_stack.html

Nationalencyklopedin. (u.d.). *Lexikalanalys*. Hämtat från Nationalencyklopedin: https://www.ne.se/uppslagsverk/encyklopedi/l%C3%A5ng/lexikalanalys

Oracle. (den 15 Februari 2015). *Chapter 2. The Structure of the Java Virtual Machine*. Hämtat från https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-2.5.3

Ruijie Fang, S. L. (den 2 November 2016). *A Performance Survey on Stack-based and Register-based Virtual.* Hämtat från https://arxiv.org/pdf/1611.00467.pdf

Sandler, N. (den 29 November 2017). *Writing a C Compiler, Part 1*. Hämtat från https://norasandler.com/2017/11/29/Write-a-Compiler.html

Stefan Marr, T. P. (den 15 September 2014). Are We There Yet? Simple Language-Implementation Techniques for the 21st Century. *IEEE Software*, ss. 60 - 67. Hämtat från https://stefan-marr.de/papers/ieee-soft-marr-et-al-are-we-there-yet/.

Tuttle, M. (2013). *fpu - printing floating points*. Hämtat från https://gist.github.com/tuttlem/4198408

Wikibooks. (den 19 Maj 2021). *x86 Assembly/NASM Syntax*. Hämtat från https://en.wikibooks.org/wiki/X86_Assembly/NASM_Syntax

Wikibooks. (den 26 September 2022). *x86 Assembly/Floating Point*. Hämtat från https://en.wikibooks.org/wiki/X86_Assembly/Floating_Point

Wikipedia. (den 25 Juli 2022). *Assembler*. Hämtat från https://sv.wikipedia.org/wiki/Assembler

Wikipedia. (den 19 November 2022). *Bytecode*. Hämtat från https://en.wikipedia.org/wiki/Bytecode

Wikipedia. (den 17 December 2022). *Machine code*. Hämtat från https://en.wikipedia.org/wiki/Machine_code

Wikipedia. (den 10 December 2022). *Stack (abstract data type)*. Hämtat från https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

Wikipedia. (den 15 December 2022). *x86 calling conventions*. Hämtat från https://en.wikipedia.org/wiki/X86_calling_conventions

8 Bilagor

8.1 Länk till GitHub repository för Ö++ projektet

https://github.com/lud99/OPlusPlus

8.2 Källkod för testprogrammen

```
srand(100);

float rand_range_float(float min, float max) => {
    return (max - min) * (to_float(rand()) / 32767.0) + min;
};

float average() => {
    float sum = 0.0;
    int its = 10000;

    for (int i = 0, i < its, i++) {
        sum += rand_range_float(0.0, 100.0);
    };

    printf("%f\n", sum);

    return sum / to_float(its);
};

print("%f\n", average())
```

Average.ö. Beräknad genomsnittet av ett stort antal slumpmässiga tal

```
float rand_range_float (float min, float max) => {
    return (max - min) * (to_float(rand()) / 32767.0) + min;
};

float pi() => {
    int inside = 0;
    int its = 10000;

    for (int i = 0, i < its, i++) {
        float x = rand_range_float(to_float(-1) * 1.0, 1.0);
        float y = rand_range_float(to_float(-1) * 1.0, 1.0);

        if (x * x + y * y <= 1.0) {
            inside++;
        };
    };

    return 4.0 * to_float(inside) / to_float(its);
};

srand(100);

print("pi = %f\n", pi());
```

pi.ö. Approximera talet pi med Monte Carlo algorithmen

```
float f(float x) => {
    return x * x * sin(x) * cos(x);
};

float derivative(float p) => {
    float h = 0.0001;

    float delta = f(p + h) - f(p);
    return delta / h;
};

for (int i = 0, i < 1000, i++) {
    float d = derivative(2.0);
};
```

Derivative.ö. Approximera derivatan för f(2)

```
float degrees_to_radians (float angle) => {
    float pi = 3.14;
    return angle * (pi / 180.0);
};

float dot(float ax, float ay, float bx, float by) => {
    return ax * bx + ay * by;
};

float dot_with_angle(float lengthA, float lengthB, float alpha) => {
    return lengthA * lengthB * cos(degrees_to_radians(alpha));
};

for (int i = 0, i < 1000, i++) {
    float a = dot(1.0, 2.0, 4.0, 5.0);
    float b = dot_with_angle(10.0, 13.0, 59.5);
};

print("dot() = %f\n", dot(1.0, 2.0, 4.0, 5.0));
print("dot_with_angle() = %f\n", dot_with_angle(10.0, 13.0, 59.5));
```

Dot.ö. Beräkna skalärprodukten av två vektorer med två olika metoder

```
int fact(int n) => {
    if (n <= 1) {
        return 1;
    };

    return n * fact(n - 1);
};

for (int i = 0, i < 1000, i++) {
    int b = fact(12);
};

print("12! = %i\n", fact(12));
```

Factorial.ö. Beräkna 12 Faktultet

```
float distance (float x1, float y1, float x2, float y2) => {
    float deltaX = abs_float(x2 - x1);
    float deltaY = abs_float(y2 - y1);

    return sqrt(pow(deltaX, 2.0) + pow(deltaY, 2.0));
};

float rand_range_float(float min, float max) => {
    return (max - min) * (to_float(rand()) / 32767.0) + min;
};

for (int i = 0, i < 1000, i++) {
    float d = distance(rand_range_float(0.0, 10.0), rand_range_float(0.0, 10.0), rand_range_float(0.0, 10.0),
rand_range_float(0.0, 10.0));
};
```

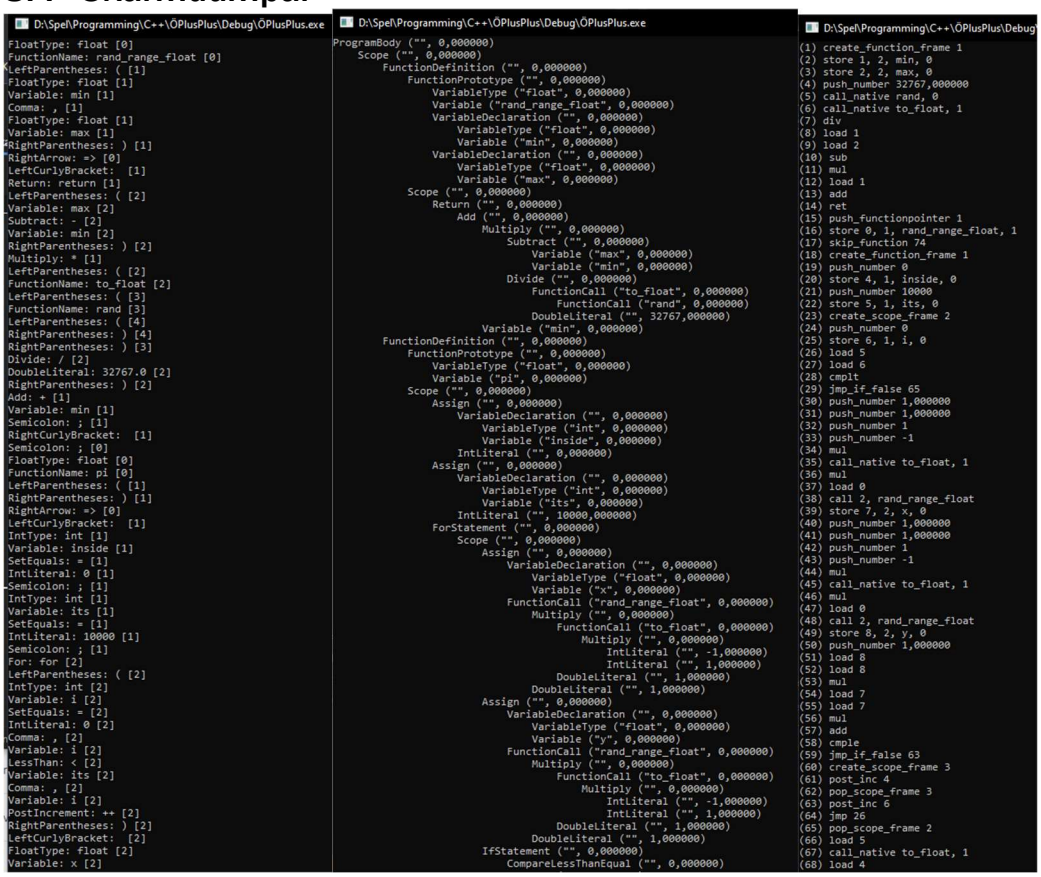
Distance.ö. Beräkna det vinkelräta avståndet mellan 2 slumpmässigt valda punkter

8.3 Tabell över mätdata

ASM						
	1	2	3	4	5	Average (ms)
average.ö	250	284	400	263	286	297
pi.ö	553	546	480	530	528	527
derivative.ö	551	462	329	402	353	419
dot.ö	272	71	229	272	478	264
factorial.ö	66	85	85	175	71	96
distance.ö	1242	1452	1410	1175	1065	1269
Bytecode						
	1	2	3	4	5	Average (ms)
average.ö	1782	1381	1580	1571	1402	1543
pi.ö	3625	3438	3173	3660	3649	3509
derivative.ö	3612	3262	3445	3797	3453	3514
dot.ö	3289	2925	3345	3360	3118	3207
factorial.ö	122	121	141	195	126	141
distance.ö	6192	5561	5582	6077	5559	5794
AST						
	1	2	3	4	5	Average (ms)
average.ö	1570	1622	1651	1631	1802	1655
pi.ö	2904	2904	3038	3023	2989	2972
derivative.ö	2975	3249	3224	3265	3079	3158
dot.ö	4168	4287	4200	4403	4301	4272
factorial.ö	78	78	113	80	104	91
distance.ö	5910	5817	6227	5934	6109	5999

Bilaga 1. Tabell över mätvärden för körningstiden vid alla fem försök

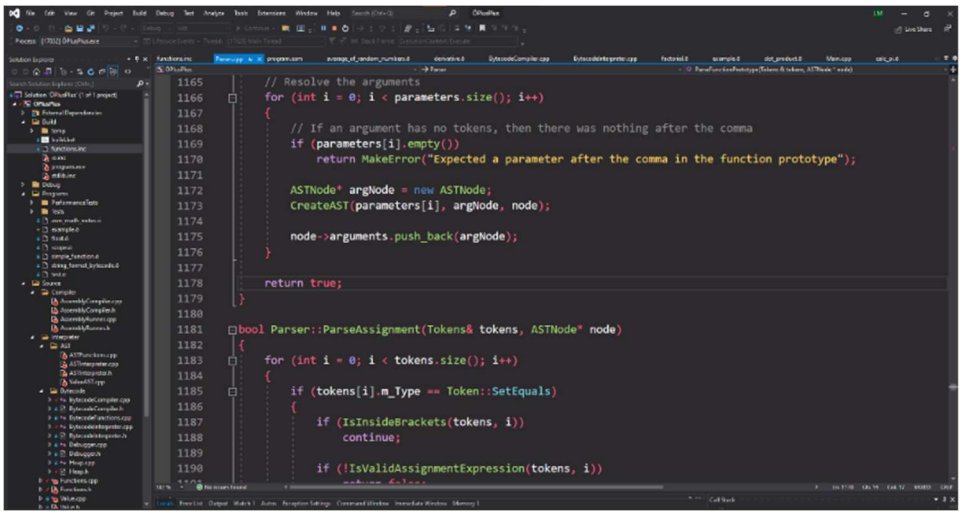
8.4 Skärmdumpar



Bilaga 2, 3 och 4. Skärmdump på vad bytecode tolkaren genererat för poletter, syntaxträd och instruktioner.

```
pow:
    ; Subroutine Prologue

    ; Create call frame
    push ebp
    mov ebp, esp

    push ebx
    ; Get arguments
    ; base
    mov eax, [ebp + 8]
    mov dword [ebp - 8], eax
    mov eax, [ebp + 12]
    mov dword [ebp - 4], eax
    sub esp, 8
    ; exponent
    mov eax, [ebp + 16]
    mov dword [ebp - 16], eax
    mov eax, [ebp + 20]
    mov dword [ebp - 12], eax
    sub esp, 8

    ; Body
    ; push rhs and lhs
    fld qword [ebp - 16] ; exponent
    fld qword [ebp - 8] ; base

    ; magic :)
    fyl2x
    fld1
    fld st1
    fprem
    f2xm1
    fadd
    fscale
    fxch st1
    fstp st0

    ; Subroutine Epilogue
    ; Restore calle-saved registers
    pop ebx
    mov esp, ebp ; deallocate local variables
    pop ebp ; restore old base pointer
    ret
```

Bilaga 5. Bild på x86 assemblykod för att beräkna x upphöjt till y



Bilaga 6. Skärmdump på projektet och utvecklingsmiljön Visual Studio 2019