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

Här skrivs en sammanfattning på engelska
Läs mer om sammanfattning här

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.

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

1.2 Metod

Arbetet utgår ifrån litteratur och vetenskapliga studier för att skapa ett programmeringsspråk grundat i principer som är standard för programmeringsspråk idag. Programmeringsspråket kommer ha möjligheten att köras genom de tre olika exekveringsmetoderna arbetet ska undersöka. Prestandan mellan metoderna kommer då att jämföras 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 inte någon påverkan på resultatet. Datorn programmen kommer testas på har en Intel Core i5-4440 @ 3.10GHz.

För att jämföra körningstiden skrivs sex olika program som utför lite matematiska beräkningar och implementerar några algoritmer. Programmet körs enligt de olika exekveringsmetoderna fem olika gånger var, där varje körning utgår 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.

Koden för att tolka det skapade språk skrivs i programmeringsspråket C++ och jag använder mig av utvecklingsmiljön Visual Studio 2019. Genererad assemblykod assembleras av den populära Netwide Assembler (NASM) och Simple ASM (SASM) används 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 assembly kod just för att C++ är närmare hårdvaran. Hade tolkaren till programmeringsspråket varit skriven i Java eller Python hade samma kod troligen gett lägre körtid. För att ge den bästa möjligheten för att rättvist jämföra exekveringsmetoderna, så används det programmeringsspråk som ger bättre prestanda.

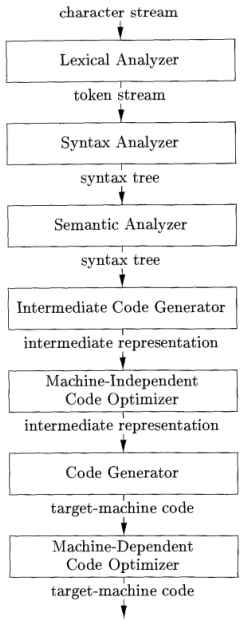
1.3 Avgränsning

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.

2 Bakgrundsinformation

2.1 Strukturen av en språkbehandlare

För att gå från källkod till något som en dator kan exekvera så krävs det ett program som ska tar in källkoden. Programmet kallas som generellast 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 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)



Figur 1. Överblick över ett exempel för strukturen av en kompilator. Tagen från ”Compilers: Principles, Techniques, and Tools”, sida 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.

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 källkoden enskilt, men med sammanhang känner igen vilka bokstäver och symboler som tillhör vilken fras, 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 ska till exempel 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. Förslagsvis sparar en token av typen *Number* också siffervärdet för numret.

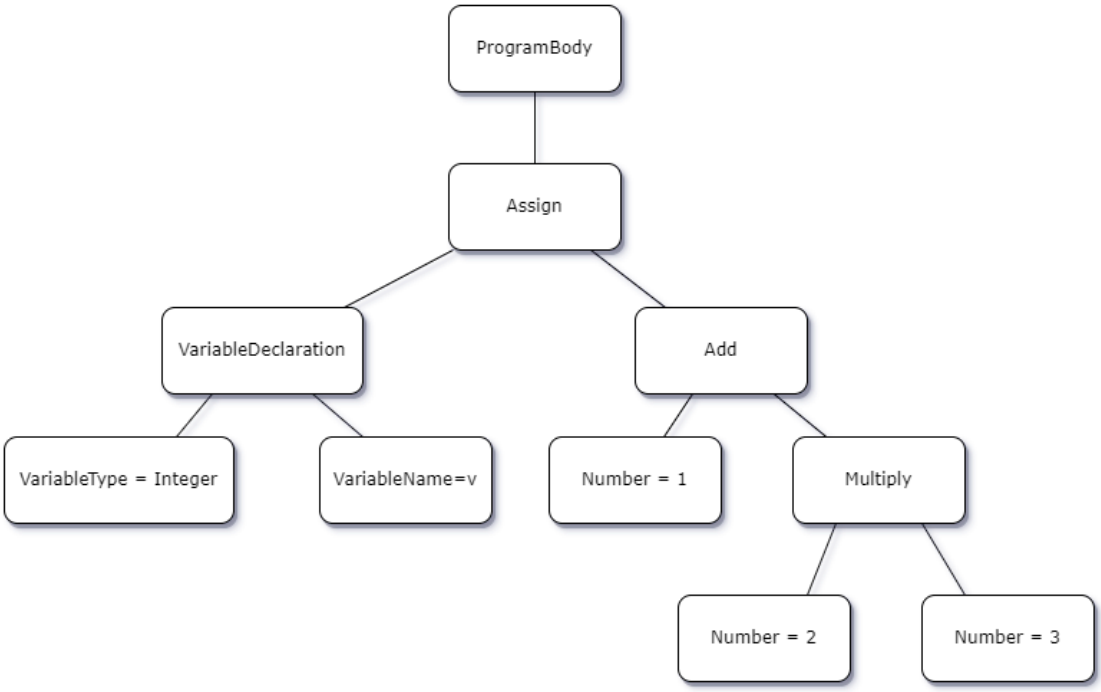
Typ	Värde
VariableType	int
Variable	v
Assign	=
Number	1
Add	+
Number	2
Multiply	*
Number	3

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

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 hierarkiella 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.



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

Ett syntaxträd är nästintill identiskt, men 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. 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.

I *figur* 3 ovan 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. Trädet innehåller tillräckligt med information om strukturen av programmet 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 bes processorn i datorn göra det så förstår den ingenting. Därför behövs kodgenereringen som skapar instruktioner processorn faktiskt förstår, i stället för en generell representation av strukturen för källkoden. Mer om detta i Kapitel 3.2 och 3.3

3 Exekveringsmetoder

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.

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. 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.

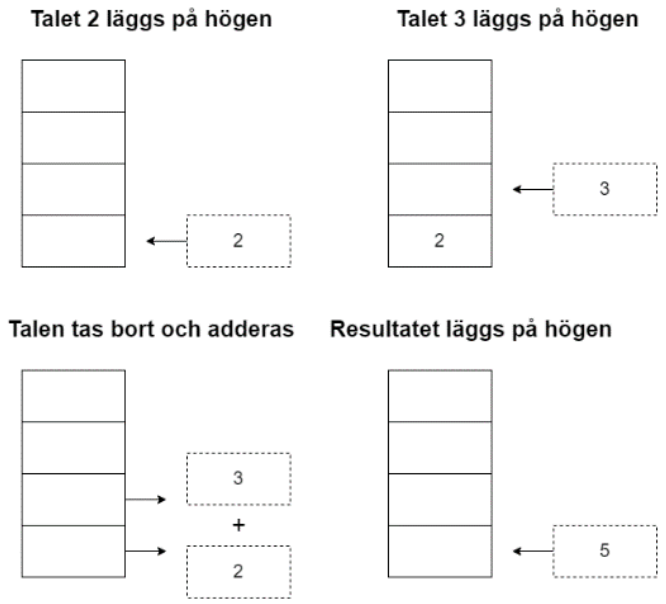
0: $push$ 2
1: $push$ 3
2: add
3: $jump$ 0
Figur 4. Exempel på ett program skrivet med bytekodinstruktioner. Programmet adderar 2 och 3 tills att programmet stängs ner

Den virtuella maskinen börjar högst upp och läser av varje instruktionerna i ordningen de är och går vidare till den nästa instruktionen 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).

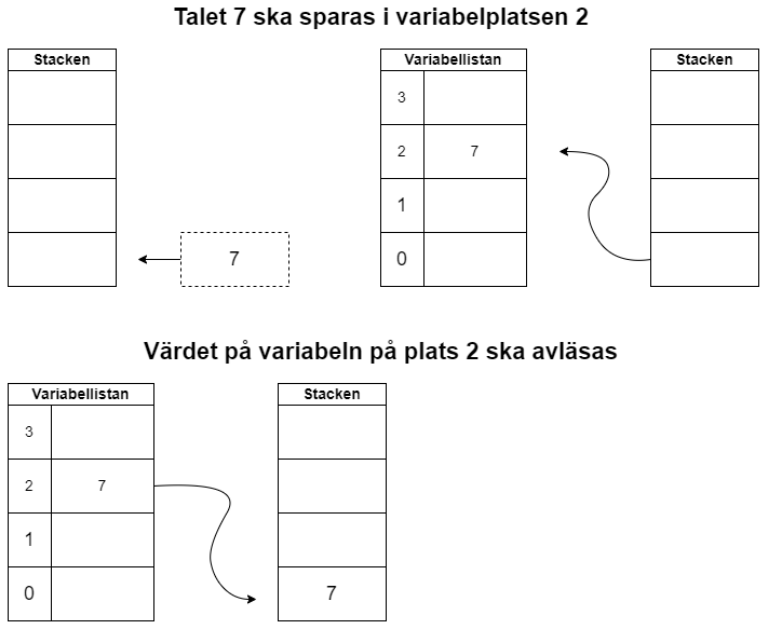
Stacken, eller högen, har sitt syfte att skicka och spara information mellan instruktioner under körning av programmet. I exempelprogrammet i *figur 4* så läggs talen 2 och 3 till i stacken. När maskinen ska tolka instruktionen *add* så vet den vad som ska adderas, för den tar ner de två värdena högst 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 *assemblyinstruktion* 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 assemblyspråket. De instruktionerna *assembleras* sedan till maskinkod bestående av endast numeriska tal, som sedan processorn kan 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. 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 vilket kod 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.

```
// 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?

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

Figur 9. Variabler av typen heltal, flyttal (double precision) och strängkonstanter.

```
global int pi = 2

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

Figur 10. Globala och lokala variabler

```
int i = 0;
i++;
i += 4;
```

Figur 11. Ökning av variabler

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

Figur 12. Matematiska operationer och tolkning av uttryck

```
float i = 0;
while (i <= 10.0) {
 if (i >= 3.9) {
    print("i >= 3.9");
 };
 i += 1.0;
}
```

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

```
if (1 == 2) {
  print("false");
} else {
  print("true");
};
```
Figur 14. Else i samband med en if-sats

```
int factorial(int n) => {
  if (n <= 1) {
    return 1;
  }
  return n * fact(n - 1);
};
```
Figur 15. Definiera, anropa egna funktioner med argument rekursivt

4.1.2 Lista över ett urval av funktioner

4.1.2.1 print

Funktionsprototyp: $void\ print(string, ...args)$
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 friheten att göra mycket av programmeringen som jag själv känt rimligt och att försöka lösa många av problemen som uppstått själv i stället för att söka hur andra implementationer gjort. Det har lett till att mina implementation skiljer sig ifrån den generella strukturen i vissa fall.

4.2.1 AST Interpreter

Tolkaren tolkar syntaxträdet direkt efter att det skapats, utan att göra djupare analyser av koden. Exempelvis, anropas en egendefinierad funktion så går tolkaren igenom noderna som om den aldrig sett den förut. En del onödig 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 och poppas. En scope frame innehåller alla variabler den 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. Sen så poppas scopet och försvinner

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 datatyperna, men även vilken av datatyp instanser 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*

```
class Value {
  int intData
  float floatData
  string stringData

  bool isGlobal;
  Enum variableType;
}

class ScopeFrame {
  hashmap<string, int> variables
}
```

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

4.2.2 Bytecode Interpreter

Efter 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. Argumenten används för att ge data som instruktionen behöver, exempelvis om ett visst värde ska läggas på högen så är det ett argument.

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, tillskillnad från den enklare variabel hashmappen i AST tolkaren. Fördelen med en statisk storlek är 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 data (se *figur 18*). En sträng allokeras och sparas i klassen Heap som kan sedan sparas i variabler och användas för funktioner genom att pekaren till strängen fungerar som ett vanligt heltal.

```
1 class Heap {
2     map<int, void*> m_Entries;
3 }
```

Figur 18. Hur klassen Heap är uppbyggd

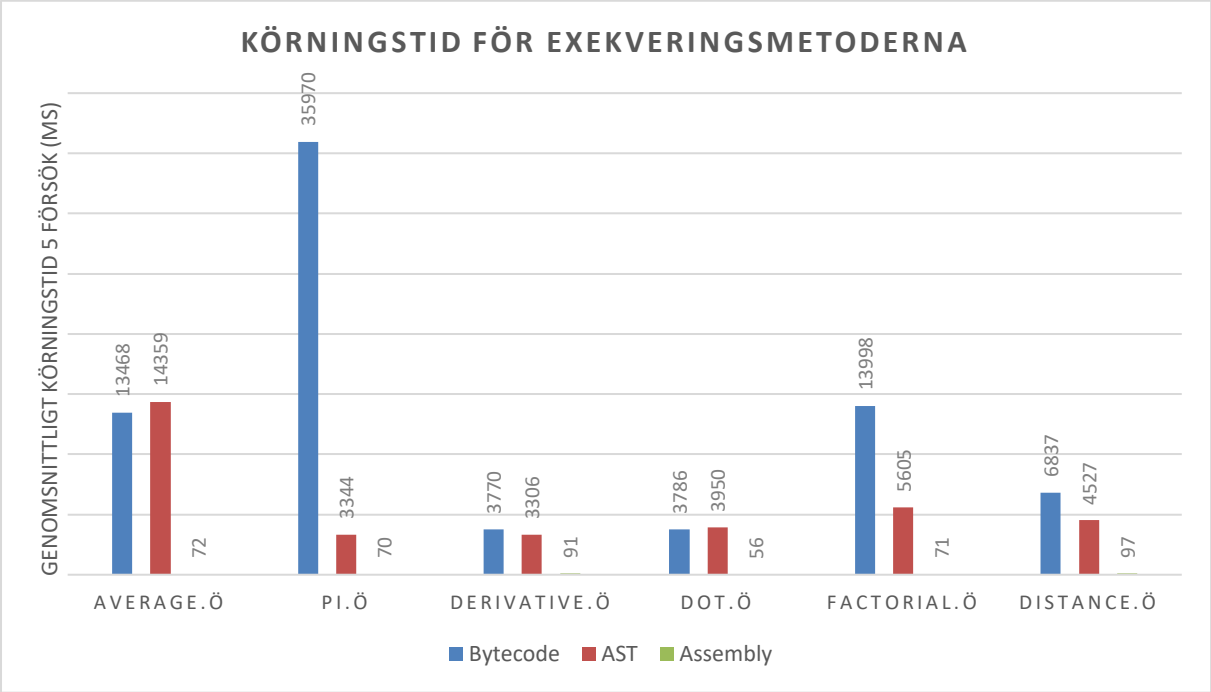
4.2.3 Assembly

Ö++ använder x86 assembly i stället för den mer moderna x64. Det är huvudsak för att det är lättare att skriva kod för en x86 processor än för x64. Det finns även mer resurser om x86 arkitekturen eftersom det har funnits mycket längre. 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, för 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 en 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 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



Som kan ses på diagrammet presterade Assemblykoden avsevärt mycket bättre än de andra två. AST och bytecode var ganska lika i prestanda i flera av testprogrammen, men speciellt *pi.ö* var flera magnituder långsammare än AST tolkaren. Bytecode presterade bättre än AST tolkaren i *average.ö*, men med ganska liten marginal.

6 Slutsatser och avslutande diskussion

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

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 Testprogram

```
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 derivatanför 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

```
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));
};
```

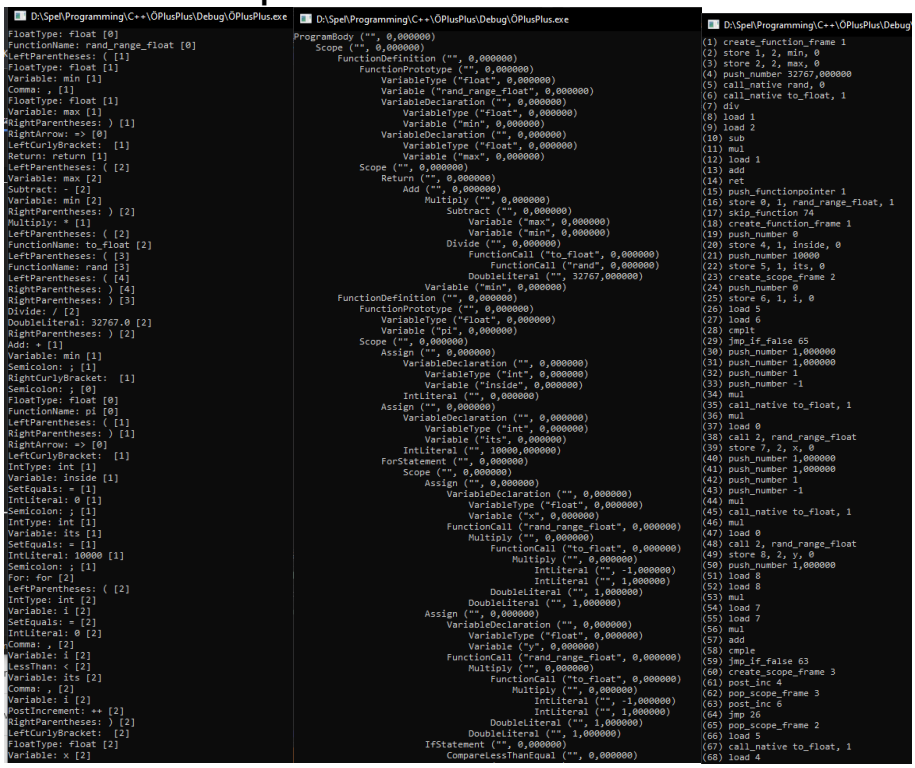
Pythagoras.ö. Beräkna det vinkelräta avståndet mellan 2 slumpmässiga punkter

8.3 Tabell över mätdata

ASM						
	1	2	3	4	5	Average (ms)
average.ö	121	71	38	91	39	72
pi.ö	55	76	63	54	101	70
derivative.ö	73	121	82	96	83	91
dot.ö	59	59	60	57	47	56
factorial.ö	99	64	65	64	62	71
distance.ö	55	50	174	66	138	97
Bytecode						
	1	2	3	4	5	Average (ms)
average.ö	13524	13489	13480	13453	13392	13468
pi.ö	36097	35823	36143	35998	35789	35970
derivative.ö	3600	3695	3948	3868	3735	3769
dot.ö	3676	3856	3698	3865	3836	3786
factorial.ö	14000	13915	14095	14005	13974	13998
distance.ö	6603	7036	6494	7167	6887	6837
AST						
	1	2	3	4	5	Average (ms)
average.ö	14569	14087	14649	14322	14169	14359
pi.ö	3650	3525	2449	3441	3655	3344
derivative.ö	3077	3457	3716	3202	3079	3306
dot.ö	3675	4171	4221	3916	3766	3950
factorial.ö	5469	5891	5568	5409	5687	5605
distance.ö	4331	4590	4596	4468	4648	4527

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

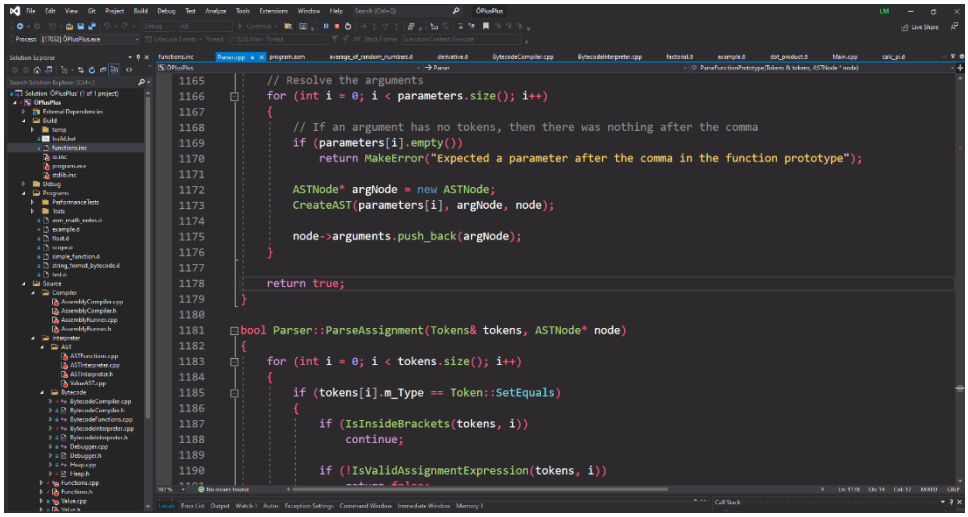
8.4 Skärmdumpar



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

```
2   pow:
3       ; Subroutine Prologue
4
5       ; Create call frame
6       push ebp
7       mov ebp, esp
8
9       push ebx
10      ; Get arguments
11      ; base
12      mov eax, [ebp + 8]
13      mov dword [ebp - 8], eax
14      mov eax, [ebp + 12]
15      mov dword [ebp - 4], eax
16      sub esp, 8
17      ; exponent
18      mov eax, [ebp + 16]
19      mov dword [ebp - 16], eax
20      mov eax, [ebp + 20]
21      mov dword [ebp - 12], eax
22      sub esp, 8
23
24      ; Body
25      ; push rhs and lhs
26      fld qword [ebp - 16] ; exponent
27      fld qword [ebp - 8] ; base
28
29      ; magic :)
30      fyl2x
31      fld1
32      fld st1
33      fprem
34      f2xm1
35      fadd
36      fscale
37      fxch st1
38      fstp st0
39
40      ; Subroutine Epilogue
41      ; Restore calle-saved registers
42      pop ebx
43      mov esp, ebp ; deallocate local variables
44      pop ebp ; restore old base pointer
45      ret
```

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



Bilaga 6. Skärmdump på projektet och utvecklingsmiljön