LINKÖPINGS UNIVERSITET

TDP019 Projekt: Datorspråk

CMM

Författare

Oliver Brandett, olibr699@student.liu.se
Anton Norman, antno464@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2024
Version 1.1

2024-05-06

Innehåll

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Dokument skapat	240503
1.1	Första utkast klart	240506

2 Inledning

Det här dokumentet skrivs som en del av det projekt som genomförs i kursen TDP019 Projekt: Datorspråk, som en del av andra terminen på programmet Innovativ programmering.

CMM som står för C Minus Minus, är ett programspråk utvecklat med inspiration från C++ och Python. Det är ett imperativt programspråk som enklast kan beskrivas som Python funktionalitet med C++ liknande syntax, vilket är varför språket har namngetts C Minus Minus, ett sämre C++ alternativ.

3 Användarhandledning

3.1 Introduktion

`CMM` är ett språk designat för de med grundläggande kunskaper av C-liknande språk så som C++ och Python. Denna handledning kommer därför att fokusera på att redogöra syntaxen för språket, snarare än hur man programmerar i CMM.

3.2 Installation

För att kunna använda CMM följer du följande instruktioner. Observera dock att detta endast fungerar på unix baserade system, såsom Linux eller MacOS.

Ladda först ner CMM's källkod genom att skriva följande kommando i din terminal:

```
git clone https://gitlab.liu.se/antno464/tdp019.git && cd tdp019
```

När du har laddat ner källkoden, behöver CMM också installeras. Detta gör du med följande kommando:

```
./install.sh
```

Nu är CMM installerat på din maskin och redo att användas. När du har skrivit din kod i en .cmm fil, och är redo att exekvera koden gör du på följande vis:

```
cmm <filnamn>.cmm
```

3.3 Huvudfunktion

All kod som exekveras i CMM måste ligga i en huvudfunktion. Denna huvudfunktion skapas genom att skriva följande:

```
void codeGoBrrrrrrrr(){
    // din kod här
}
```

Mängden 'r' i deklarationen är inte strikt bestämd, det ända kravet är att det finns minst två stycken 'r'. Detta är för att ge användaren friheten att uttrycka sin entusiasm för den kod som de har skrivit. Huvudfunktionen måste vara av typen void, den måste även ligga längst ner i CMM-filen. Alla andra funktionsdeklarationen måste därför ligga ovanför huvudfunktionen, detta är eftersom att CMM läser koden uppifrån och ner och behöver då ha alla funktioner definierade innan dess att de kallas på i huvudfunktionen.

3.4 Funktioner

Deklarationen för en funktion är likadan som den för huvudfunktionen förutom det att funktionens namn behöver följa två regler; namnet måste börja på en liten bokstav och namnet får inte innehålla några siffror. Det gäller även att se till att det finns en retur någonstans i funktionens kod, med undantag för funktioner av typen void. För att använda sig av funktionsparametrar så gäller det att man skriver av vilken typ som parametern skall vara samt ett variabelnamn, läs mer om variabler under 'Variabeltilldelningar' nedan.

Här är ett exempel på en enkel funktion:

```
int fun(int x){
    return x;
}
```

En funktion kan kallas direkt eller via en variabeltilldelning eller som en del av ett uttryck i en if-sats. Exempel på funktionsanrop:

```
void codeGobrrrr(){
    fun(5); // direkt anrop av funktionen
    int a = fun(10); // funktionenes returvärde tilldelas variabeln a
    if(fun(7) == 7){ // funktionens returvärde används i jämförelsen
        //...
    }
}
```

3.5 Variabeltilldelningar

Namngivningen av variabler följer samma regler som för funktioner, det vill säga, namnet börjar med en liten bokstav och innehåller inga siffror. För att skapa en variabel så måste man samtidigt tilldela den ett värde enligt följande syntax:

```
int var = 10;
```

För att ändra värdet på en redan existerande variabel så skriver man väldigt likt koden ovan men man har inte med typen då variabeln är skapad och då vet vilken typ värdet den lagrar skall vara. Se därför till att du tilldelar ett värde av rätt typ, annars blir det fel och koden kommer inte exekveras.

3.6 If-satser

If-satser i CMM är mycket likt if-satser i andra språk. Nyckelordet för att initiera en sats är if, följt av parenteser som innehåller det uttryck som ska utvärderas. Därefter kommer det kodblock som ska köras. If-satser kan även kedjas ihop med flera else-if grenar, samt en else gren på slutet. Även dessa fungerar som de gör i många andra liknande språk. Se exempelkod nedan:

```
if(a <= 10 && b != 11) {
    // din kod här
```

```
}
elif(c != False) {
    // din kod här
}
else {
    // din kod här
}
```

3.7 Loopar

I CMM finns både while- och for-loopar. Båda dessa är återigen väldigt likt andra C baserade språk. Nyckelorden för att initiera dessa är while eller for. En while loop kräver endast ett uttryck som utvärderas inför körning. En for loop kräver utöver detta uttryck även en styrvariabel, samt ett inkrement som uppdaterar värdet på denna styrvariabel. Se exempel på dessa nedan:

```
while(i < 25) {
    // din kod här
}
...
for(int j = 0; j < 10; j++) {
    // din kod här
}
```

3.8 In- & Out-put

I CMM finns stöd för att skriva ut information till terminalen, samt att tillåta användaren att mata in information från terminalen. För att skriva ut något till terminalen använder du nyckelordet print. Du använder det på följande sätt:

```
print("Detta skrivs ut till terminalen");
```

Om du istället vill ta in något från terminalen använder du nyckelordet input. Detta används på följande sätt:

```
print("Skriv en siffra: ");
int var = input();
```

3.9 Strings & Arrays

CMM har stöd för vissa sammansatta datastrukturer, nämnligen strings och arrays. Båda dessa är implementerade med en enkel-länkad nodlista. Dessa kommer även med vissa medlemsfunktioner såsom en index operator med mera. För att skapa en string är det på samma sätt som andra variabler, men med typen string. För att skapa en array måste man även specificera vilken data-typ som ska lagras. Du skapar båda dessa på genom att skriva följande kod:

```
string text = "Detta sparas som en string";
array<int> lista = [1, 2, 3, 4];
```

4 Systemdokumentation

CMM är byggt ovanpå Ruby och använder sig av dess interpretator för att översätta CMM kod till maskinkod som en dator kan exekvera. All CMM kod parsas och lexas av 'rdparse', en recursive decent parser som var tillhandhållen för utvecklingen av språket.

4.1 Grammatik/BNF

⟨program⟩ ::= ⟨func_declares⟩

⟨func_declares⟩ ::= ⟨func_declares⟩⟨func_declare⟩ | ⟨func_declare⟩

⟨func_declare⟩ ::= ⟨main⟩ | ⟨new_function⟩

⟨new_function⟩ ::= ⟨type⟩⟨identifier⟩`(`⟨params⟩`){`⟨code_blocks⟩`}` | ⟨type⟩⟨identifier⟩`( ){`⟨code_blocks⟩`}`

⟨main⟩ ::= `void /codeGobrr[r]+/ ( ) { }` | `void /codeGobrr[r]+/ ( ) {`⟨code_blocks⟩`}`

⟨code_blocks⟩ ::= ⟨code_blocks⟩⟨code_block⟩ | ⟨code_block⟩

⟨code_block⟩ ::= ⟨increment⟩`;` | ⟨decrement⟩`;` | ⟨multiment⟩`;` | ⟨diviment⟩`;` | ⟨potensiment⟩`;` | ⟨list_opp⟩`;` | ⟨string_opp⟩`;` | ⟨input⟩`;` | ⟨assign⟩`;` | ⟨print⟩`;` | ⟨if_statement⟩ | ⟨change_var⟩ | ⟨while_loop⟩ ⟨for_loop⟩ | ⟨call_function⟩`;` | ⟨break⟩ | ⟨return⟩`;`

⟨input⟩ ::= `input ( )`

⟨call_function⟩ ::= ⟨identifier⟩`(`⟨values⟩`)` | ⟨identifier⟩`( )`

⟨return⟩ ::= `return`⟨value⟩ | `return`⟨if_expr⟩ | `return`

⟨assign⟩ ::= ⟨type⟩⟨identifier⟩`= input ( )` | ⟨type⟩⟨identifier⟩`=`⟨call_function⟩ | ⟨type⟩⟨identifier⟩`=`⟨value⟩ | `array <`⟨type⟩`>`⟨identifier⟩`=`⟨value⟩ | `array <`⟨type⟩`>`⟨identifier⟩`= [ ]`

⟨list_opp⟩ ::= ⟨identifier⟩`[`⟨index_opp⟩⟨]⟩`=`⟨value⟩ | ⟨identifier⟩`[`⟨index_opp⟩`]` | ⟨identifier⟩`.append(`⟨value⟩`)` | ⟨identifier⟩`.pop()` | ⟨identifier⟩`.size()` | ⟨identifier⟩`.conga(`⟨value⟩`)` |

⟨index_opp⟩ ::= ⟨Integer⟩ | ⟨value⟩

⟨string_opp⟩ ::= ⟨indetifier⟩`[`⟨index_opp⟩`]` | ⟨indentifier⟩`.size()` | ⟨indetifier⟩`.conga(`⟨value⟩`)`

⟨change_var⟩ ::= ⟨identifier⟩`=`⟨value⟩`;`

⟨if_statement⟩ ::= `if(`⟨if_expr⟩`){`⟨code_blocks⟩`}`⟨else_statement⟩ |

		`if(`⟨if_expr⟩`){`⟨code_blocks⟩`}`⟨elifse_statements⟩ |
		`if(`⟨if_expr⟩`){`⟨code_blocks⟩`}`
⟨if_expr⟩	::=	⟨value⟩⟨operator⟩⟨value⟩ | ⟨value⟩
⟨else_statement⟩	::=	`else{`⟨code_blocks⟩`}`
⟨elif_statements⟩	::=	⟨elif_statements⟩⟨elif_statement⟩ | ⟨elif_statement⟩
⟨elif_statement⟩	::=	`elif(`⟨if_expr⟩`){`⟨code_blocks⟩`}`⟨elifse_statement⟩ |
		`elif(`⟨if_expr⟩`){`⟨code_blocks⟩`}`⟨else_statement⟩ |
		`elif(`⟨if_expr⟩`){`⟨code_blocks⟩`}`
⟨while_loop⟩	::=	`while(`⟨if_expr⟩`){`⟨code_blocks⟩`}`
⟨for_loop⟩	::=	`for(`⟨assign⟩`;`⟨if_expr⟩`;`⟨increment⟩`){`⟨code_blocks⟩`}` |
		`for(`⟨assign⟩`;`⟨if_expr⟩`;`⟨decrement⟩`){`⟨code_blocks⟩`}`
⟨break⟩	::=	`break;`
⟨increment⟩	::=	⟨value⟩`++` | ⟨value⟩`+=`⟨num⟩
⟨decrement⟩	::=	⟨value⟩`-` | ⟨value⟩`-=`⟨num⟩
⟨multiment⟩	::=	⟨value⟩`*=`⟨num⟩
⟨diviment⟩	::=	⟨value⟩`/=`⟨num⟩
⟨multiment⟩	::=	⟨value⟩`**=`⟨num⟩
⟨operator⟩	::=	`==` | `!=` | `<=` | `>=` | `<` | `>` | `%` | `&&` | `||`
⟨print⟩	::=	`print(\\);` | `print();` | `print(`⟨value⟩`);`
⟨params⟩	::=	⟨params⟩`,`⟨haram⟩ | ⟨haram⟩ |
⟨haram⟩	::=	⟨type⟩⟨identifier⟩
⟨type⟩	::=	`int` | `float` | `bool` | `char` | `string` | `void`
⟨values⟩	::=	⟨values⟩`,`⟨value⟩ | ⟨value⟩

⟨value⟩ ::= ⟨boolean⟩ | ⟨comp⟩ | `[`⟨comp⟩`]`

⟨boolean⟩ ::= `True` | `False`

⟨identifier⟩ ::= `/[a-z]+[\w]*/`

⟨comps⟩ ::= ⟨comps⟩`,`⟨comp⟩ | ⟨comp⟩

⟨comp⟩ ::= ⟨comp⟩`<`⟨expr⟩ | ⟨comp⟩`>`⟨expr⟩ | ⟨comp⟩`<=`⟨expr⟩ | ⟨comp⟩`>=`⟨expr⟩ | ⟨comp⟩`==`⟨expr⟩ | ⟨comp⟩`!=`⟨expr⟩ | ⟨comp⟩`&&`⟨expr⟩ | ⟨comp⟩`||`⟨expr⟩ | ⟨expr⟩

⟨expr⟩ ::= ⟨expr⟩`+`⟨term⟩ | ⟨expr⟩`-`⟨term⟩ | ⟨term⟩

⟨term⟩ ::= ⟨term⟩`*`⟨expo⟩ | ⟨term⟩`/`⟨expo⟩ | ⟨term⟩`%`⟨expo⟩ | ⟨expo⟩

⟨expo⟩ ::= ⟨term⟩`*`⟨expo⟩ | ⟨expo⟩`**`⟨expo⟩ | ⟨num⟩

⟨num⟩ ::= ⟨call_function⟩ | `/[0-9]*\.[0-9]+/` | `/\d+/` | `-/\d+/` | `-/\d+/./\d+/` | ⟨list_opp⟩ | ⟨identifier⟩ | `'/\w{1}/'` | `\`⟨string⟩`\` | `(`⟨comp⟩`)`

⟨string⟩ ::= ⟨string⟩⟨string_part⟩ | ⟨string_part⟩

⟨string_part⟩ ::= | `/[^\s"]+/`

4.2 Parser, Lexer

CMM använder en lexer och en parser som båda är en del av den givna parsern rdparse. Lexern går igenom hela filen uppifrån och ner och gör om de tecken den hittar till mer användbara tokens. Den översätter alltså det den hittar till något som parsern sedan kan förstå. Parserns jobb är att använda dessa tokens den fått från lexern och bygga upp ett abstrakt syntax träd (AST) av det, som består av olika typer av noder. När hela trädet med alla noder är uppbyggt exekveras allting, och programmet körs.

4.3 Klasser

CMM's AST är uppbyggt av en mängd olika noder som alla har olika uppdrag och ändamål. Dessa noder är vad som ger språket, och programmet skriver i språket, sin funktionalitet. Dessa noder förklaras mer i detalj i de kommande underrubrikerna.

4.3.1 Node

Node är den klassen som alla andra klasser ärver ifrån. Vi har valt denna arvstruktur med avsikt att få allt mer sammahängande och enklare att hantera. Node klassen består endast av en tom evaluate funktion och en tom konstruktor. Dessa skrivs över av andra klasser längre ned i arvskedjan.

4.3.2 Expression Node

Expression Node ärver direkt från Node, men tillför inga direkta funktionaliteter. Den har en tom evaluate funktion och en tom konstruktor, vilket funktionellt gör den likvärdig med Node. Den används dock för att koppla samman andra noder av typen expression. Dessa är Arithmetic Expression, Binary Expression och Unary Expression.

4.3.3 Arithmetic Expression

Som namnet antyder används denna för att utföra aritmetiska uträkningar, det vill säga matematik. Dess konstruktor består av ett högerled, vänsterled och en operator som utgör beräkningen. Evaluate funktionen gör om dessa attribut till något den kan använda och utför sedan beräkningen.

4.3.4 Binary Expression

Denna nod används för att utföra binära beräkningar, det vill säga sådana uttryck där resultatet är antingen sant eller falskt. Dess konstruktor är identisk med Arithmetic Expression, och dess Evaluate funktion är mycket lik också. Evaluate funktionen returnerar dock ett booléiskt värde till skillnad mot Arithmetic Expression.

4.3.5 Unary Expression

Denna nod används för att beräkna det booléiska värdet av endast en term. Dess konstruktor består enbart av en term, och evaluate funktionen gör om denna term till något den kan använda för att sedan enbart returnera det värdet.

4.3.6 CmmInteger

CmmInteger är CMM's implementation av integers, det vill säga heltal, i CMM. Dess konstruktor sätter endast värdet på objektet, och dess evaluate returnerar det i en form som parsern kan använda.

4.3.7 CmmFloat

CmmFloat är CMM's implementation av flyttal, med andra ord ett tal med decimaler. Likt CmmInteger sätter konstruktorn endast värdet på objektet och evaluate funktionen returnerar det värdet på ett användbart sätt.

4.3.8 CmmBoolean

CmmBoolean är CMM's implementation av booléiska värden, det vill säga ett värde som är antingen sant eller falskt. Likt CmmInteger och CmmFloat sätter konstruktorn endast värdet på objektet och evaluate funktionen returnerar det värdet på ett användbart sätt.

4.3.9 CmmChar

CmmChar är CMM's implementation av chars, det vill säga enstaka karaktärer. Konstruktorn sätter värdet på den, och evaluate funktionen returnerar värdet på ett användbart sätt.

4.3.10 CmmString

CmmString är CMM's implementation av strängar. Konstruktorn får in in ruby-string som parameter, men delar sedan upp den till en lista av dess beståndsdelar och skapar en instans av en länkad lista som används under huven för att representera denna sträng. Dess evaluate funktion returnerar en sammansatt sträng, uppbyggd av de elementen som finns i listan.

4.3.11 Print

Print noden används för att skriva ut saker till terminalen och därmed ha någon form av utdata ström i CMM. Dess konstruktor får in ett värde i form av en godtycklig CMM nod. I evaluate funktionen bryts denna nod sedan ned till något som kan användas av den inbyggda ruby funktionen puts, som används för att skriva ut till terminalen.

4.3.12 CmmInput

Input noden är motsatsen till print noden. Den används för att få in data från användaren via terminalen. Dess konstruktor tar endast in en datatyp i from av en sträng. I evaluate funktionen används sedan den inbyggda ruby funktionen stdin.gets för att få in data från terminalen, lagra den i en temporär variabel, för att slutligen använda denna datatyp-variabel för att säkerställa att det som matas in är av den datatyp som det ska vara. Skulle det inte stämma överens, avbryts programmet med ett felmeddelande. Om det däremot stämmer överens returneras det inmatade värdet i form av en motsvarande CMM datatyp.

4.3.13 Assignment

Assignment noden ansvarar för att tilldela variabler. Dess konstruktor tar in en identifier (variabelnamn), ett värde och en datatyp. Assingment nodens evaluate funktion kollar vilken datatyp noden har och beroende på vad datatypen är, kallas en tredje funktion med motsvarande parameter. Den funktionen kollar så att datatypen som skickas in, stämmer överens med den angivna. Om det stämmer sätts variabeln i lagring, annars avbryts körningen och ett felmeddelande skickas.

4.3.14 Look Up

Look Up nodens konstruktor tar en sträng som parameter som ska motsvara ett variabelnamn. Evaluate funktionen använder denna sträng, för att kolla i lagringen om den finns. Om den hittas returneras dess värde, om inte skrivs ett felmeddelande.

4.3.15 Change Var

Change Var noden fungerar likt Look Up noden. Den skiljer sig i det att dess konstruktor har en ytterligare parameter – nytt värde – och att när variabeln i fråga har hittats returneras inte dess värde, utan ändras direkt på plats i lagringen.

4.3.16 List Node

List noden används för att representera en lista. Den har värden och datatyp som attribut. Under huven är den lik CmmString i den mån att den har en länkad lista som underliggande datastruktur. Den har ett flertal hjälpfunktioner som bidrar med olika saker, främst när listan skapas. Dess evaluate funktion kollar endast om listan är genererad och returnerar en representation av listan i strängformat. De andra funktionerna är hjälper till med att kontrollera datatyperna på alla element som skickas in i listan.

4.3.17 List Opp

List Opp skulle kunna ses som en hjälpklass till List Node. Den innehåller ett flertal operationer som kan utföras på en lista. Den tar in en Look Up nod som parameter till sin konstruktor, samt vissa andra parametrar som avgör vilken operation som ska utföras.

4.3.18 String Opp

String Opp fungerar på samma sätt som List Opp till listor, men för strängar. Då båda har samma länkade lista som underliggande datastruktur, använder den samma parametrar som List Opp. Det som skiljer dem

åt är antalet och vilka funktionaliteter som tillåts.

4.3.19 Container Node

Container Node är en mellan klass mellan de noder som innehåller ett eget lokalt scope, och basklassen Node. Alla styrsstrukturer som kan ha lokala instruktioner som ska utföras när vissa krav uppfylls, ärver från Container Node. Har inga parametra, men vissa extra funktioner som alla härledda klasser har. Vissa använder basversionen av dessa funktioner, medans andra behöver lite tillägg och skriver i de fall över den befintliga. Dessa funktioner hanterar retursatser angående funktioner. Funktionerna är has_return och will_return som kollar om, respektive vad som kommer returneras.

4.3.20 If Statement

If Statement ärver från Container Node och har därmed de två extra funktionerna som Container Node medför. If Statement har två attribut och parametrar till sin konstruktor. Dessa är ett uttryck i form av en expression node, och en lista över instruktioner som ska utföras om uttrycket stämmer. Dess evaluate funktion kollar helt enkelt om uttrycket stämmer, och utför instruktionerna i listan om det skulle stämma.

4.3.21 Elif Statement

Elif Statement ärver från Container Node, men utökar de två extra funktionerna något. Elif noden har tre attribut som skickas in till konstruktorn – uttrycket, lista över instruktioner och en nästkommande elif-sats. De två förstnämnda är detsamma som för If noden, men elif satser kan även kedjas och därför behöver den hålla koll på vilken sats som kommer härnäst. Denna kan vara antingen en till elif nod, eller en else nod. Elif nodens evaluate funktion fungerar väldigt likt is nodens, men den kallar på dess nästa elif-nods evaluate funktion om inte det nuvarande uttrycket skulle stämma.

4.3.22 Else Statement

Else Statement ärver från Container Node, och skriver över en av dess funktioner – will_return. Else noden har endast ett attribut som skickas in till konstruktorn vilket är dess lista över instruktioner. Else nodens evaluate funktion går endast genom den listan och utför alla instruktioner i den.

4.3.23 While Loop

While Loop noden ärver från Container Node. Den tar in ett uttryck som styr huruvida körningen ska fortgå eller ej. Den har också en lista över instruktioner som ska utföras om uttrycket sämmer. Dess evaluate funktion kör endast dessa instruktioner tills dess att uttrycket inte längre uppnås, eller loopen bryts.

4.3.24 For Loop

For Loop noden ärver från Container Node. Den är väldigt lik while loopen, men skiljer sig i det faktum att den har en inbyggd styrvariabel, och ett inkrement/dekrement längst bak i sin lista över instruktioner. Detta inkrement/dekrement uppdaterar denna styrvariabel efter varje varv i körningen, vilket tillslut kommer göra att uttrycket utvärderas till falskt och loopen avbryts. For loopens evaluate funktion är näst intill identiskt till while loopen, bortsett från att denna styrvariabel skapas innan loopningen sker, och förstörs när den är klar.

4.3.25 CmmBreak

CmmBreak har inga attribut kopplade till sig utan fungerar endast som en brytpunkt i en loop. Dess evaluate funktion sätter en variabel till sant, som sedan kan hittas av loop noderna, för att veta att loopningen ska avbrytas. Denna variabel har ett namn som inte tillåts enligt variabeltilldelningen vilket innebär att den inte kommer kunna skrivas över.

4.3.26 Crement Noder

Crement noder innefattar alla de noder som direkt ändrar värdet på ett heltal. De crement som finns i CMM är de vanliga inkrement och dekrement, men även ett antal fler udda: multiment, diviment, potensiment. Multiment multiplicerar det första talet med det andra. Diviment delar det första talet med det andra. Potensiment höjer upp det första talet med det andra. Se exempel på detta nedan:

```
int x = 2;
x += 3; // => 5
x -= 3; // => -1
x *= 3; // => 6
x /= 2; // => 1
x **= 3; // => 8
```

4.3.27 CmmFunction

CmmFunction är den klassen som används för att representera funktioner i CMM. Dess attribut som skickas in till konstruktorn är: den datatyp funktionen ska returnera, de parametrar som funktionen tar in och de instruktioner som ska utföras i funktionen. CmmFuncion's evaluate funktion tar in en parameter som är de parametrar som skickas in när funktionen i fråga anropas. Det första den gör är att den tilldelar alla dessa parametrar på ett sätt som gör de användbara av de instruktioner som ska utföras i funktionen, samt kollar så att alla parametrar är av korrekt datatyp. Skulle någon av de angivna parametrarna inte stämma överens med de som krävs, kommer ett felmeddelande visas. När alla parametrar har tilldelats utförs alla instruktioner i funktionen.

4.3.28 CmmReturn

CmmReturn är en klass som hanterar returer ut ur funktioner. CmmReturn's attribut som skickas in till konstruktorn är den datatypen som den behöver vara och det som ska returneras. Det finns även ett attribut vid namn reset_thing som används för att återställa CmmReturn'en efter det att den har använts. I dess evaluate funktion kollas datatypen så att allt stämmer överens med vad som ska returneras, och sedan endast returnera det. När CmmFunction plockar upp denna retur, returnerar även denna samma sak och på så vis avslutar körningen av funktionen.

4.3.29 Function Call

Function Call används för att skapa ett funktionsanrop. Den tar in en sträng som används för att hitta rätt funktion, och en lista över de parametrar som ska användas i funktionen. Denna lista består av ett godtyckligt antal hashmappar med parameternamn och värde. I evaluate funktionen letas funktionen med det angivna namnet upp, för att sedan kalla på evaluate funktionen, med parametrarna, i CmmFunction objektet med motsvarande namn.

4.3.30 Function Declaration

Function Declaration noden används för att skapa en ny CmmFunction och lagra den på ett sätt som gör den användbar senare. Den tar in en sträng för funktionsnamnet och ett färdigt CmmFunction objekt som redan innehåller allt den behöver. I evaluate funktionen sätts funktionen endast i lagring för att senare kunna användas senare.

4.3.31 Main Function

Main Function är där hela programmet körs ifrån. Den tar endast in en lista över alla instruktioner som ska utföras. I evaluate funktionen går den endast igenom denna lista och kallar evaluate på alla instruktioner i

listan.

4.4 Tokens

Tokens skapas utifrån regexp matchningar av den text som parsern plockar upp. De tokens som skapas går sedan igenom de regler som har satts upp för att fånga vissa specifika följder av ord och tecken, om en regel matchas korrekt så skapas ett klassobjekt som sedan vid runtime kommer att exekvera kod i Ruby som representerar vad den CMM koden som skickades in uttryckte.

4.5 Kodstandard

CMM följer en väldigt avslappnad kodstandard, rader avslutas med semikolon (;) och nya scopes öppnas med klammerparanteser ({}). CMM tar ingen hänsyn till indentering så användare kan välja att skriva sin kod med eller utan beroende på preferens.

5 Programkoden

Länk till GitLab repo: https://gitlab.liu.se/antno464/tdp019

6 Reflektion

Projektet i sin helhet har varit mycket givande och roligt att utföra, då vi har fått testa på nya tekniker och lärt oss mer grundläggande hur programmeringsspråk funkar. Detta har gett oss en bättre förståelse och uppskattning för de språk vi tidigare har använt. Vi har däremot märkt hur mycket vi uppskattar när det finns en tydligt standard och tydligt uppsatta regler för hur ett språk bör användas, genom att använda Ruby. Ruby är väldigt löst i den aspekten, vilket vi uppfattade som osammanhängande och ibland till och med lite diffust. Det är däremot väldigt förlåtande i den mån att det tillåter näst intill allting som giltig syntax.

Vi stötte på vissa svårigheter på vägen. Detta var främst gällande hur språket skulle hanter scopes på ett korrekt sätt. Vi hade bekymmer med att få saker att lägga sig på rätt ställen och att göra vissa variabler skulle kommas åt från rätt ställen, och sedan förstöras när de inte längre skulle existera. Vi har även haft bekymmer med att få rekursiva funktioner att fungera helt som det ska. De fungerar nu på en grundlig nivå nu, men fortfarande inte fullständigt. Resterande delar gick relativt smidigt att implementera då vi ofta hade en ganska bra idé och tanke över hur det skulle fungera i praktiken. Självklart gick vissa delar enklare än andra, men ingenting utöver det ovannämnda var extremt avancerat. Det hände vid ett flertal tillfällen att efter vi lade till ny funktionalitet, hade befintlig funktionalitet påverkats och slutat fungera. Detta nya fel gick dock ganska fort att söka upp och lösa då dessa problem nästan alltid var att vi hade missat att framtidssäkra den äldre funktionaliteten.

Den största skillnaden mellan vår språkspecifikation och den slutliga produkten är att vi inte har implementerat objektorientering. Denna skillnaden var helt enkelt på grund av tidsbrist då vi underskattade hur lång tid alla imperativa delar av språket skulle ta. En annan skillnad mellan specifikation och produkt är att vi nämnde i specifikationen att semikolon skulle vara valfria att använda, men i den slutliga produkten måste de användas på varje rad. Detta är av den enkla anledningen att vi ville hålla oss till en strikt syntax och därmed göra CMM kod mer läsbar, och inte minst lättare att hitta fel i.