LINKÖPINGS UNIVERSITET

TDP019 Projekt: Datorspråk

CTML++

Författare

Behrad Behnoushan, behbe027@student.liu.se
Jakob Norberg, jakno825@student.liu.se

Linköpings Universitet
Språkdokumentation för kursen TDP019 – Projekt: Datorspråk

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2025
Version 1.1

2025-06-02

Innehållsförteckning

1 Inledning

Detta dokument beskriver ett projekt som genomförs under den andra terminen av programmet Innovativ Programmering vid Linköpings Universitet, som en del av kursen TDP019 – Projekt: Datorspråk. Efter avslutad kurs förväntas vi uppnå följande[1]:

- konstruera ett mindre datorspråk
- diskutera och motivera designval i det egna datorspråket med utgångspunkt i teori och egna erfarenheter
- implementera verktyg (interpretator, kompilator, etc) för det egna datorspråk
- formulera teknisk dokumentation av det egna datorspråket
- presentera en uppgift muntligt inför publik enligt angivna förutsättningar

Datorspråket vi har skapat inom projektet är CTML++, som tar stark inspiration från C++ och HTML i både syntax och språkegenskaper. CTML++ bygger på Ruby och är ett dynamiskt typat språk med stark typbundenhet, vilket innebär att konstruktioner är bundna till sina deklarerade datatyper. Vidare har språket statisk scope hantering.

[1] Lärandemål

2 Användarhandledning

I det följande avsnittet förklaras hur CTML++ installeras, vilken målgrupp språket riktar sig till samt hur det används.

2.1 Installation

Ladda ner ruby på datorn genom att följa ruby's installationsmanual[2]. Ladda sedan ner filerna tillhörande projektet[3]. Skriv *cd* i terminalen för att komma till root-katalogen. Nu kan du skriva *cd tdp019* i terminalen för att förflytta dig till projekt-katalogen. Väl i projekt-katalogen skriv *ruby parser.rb* i terminalen. CTML++ körs nu i interaktivt läge. För att köra en fil placerad i samma katalog eller under, skriv *ruby parser.rb <filnamn>*.

Tester för spårket ligger under mappen *tests*, och för att köra alla tester samtidigt finns ruby filen *allTests.rb* som kan köras enlight instruktionerna ovan.

2.2 Målgrupp

Detta dokument riktar sig till användare med grundläggande kunskaper i programmering, särskilt i språket C++. Målgruppen förväntas ha förståelse för variabeldeklaration. Viss erfarenhet av HTML och C++ kan underlätta förståelsen av CTML++ syntaxen, men är inte ett krav. Syntaxen är lätt att ta till sig genom de exempel som presenteras i dokumentet.

2.3 Kodkonstruktioner

I vissa kodexempel används → för att visa värdet av ett uttryck, detta är inget som används i den faktiska koden utan endast för demonstration. Vissa kodexempel innehåller även indentering, det vill säga att delar av koden är inflyttad från vänstra sidan. Detta är inget CTML++ tar hänsyn till utan används endast för enklare läsning. Det viktiga i CTML++ är att du stänger kodblock med *</**block**>* där **block** ersätts med det **block** du är i.

2.3.1 Datatyper

I CTML++ representeras datatyper av objekt, varav det finns det 4 inbyggda:

1. Integer, representeras av nyckelordet *int*: Positiva eller negativa heltal.
2. Float, representeras av nyckelordet *float*: Decimaltal, det vill säga tal som innehåller en punkt, till exempel 1.5 eller 2.0.
3. Boolean, representeras av nyckelordet *bool*: Booleska värden är antingen sanna eller falska. Bools i CTML++ är inte sina egna objekt, utan representeras istället av heltal - där 0 motsvarar falskt och alla andra tal motsvarar sant, sant och falskt omvandlas i sin tur till 1 och 0.
4. LList, representeras av nyckelordet *llist*: Länkad lista med svagt typade element utan typkontroll. Stödjer *random access* med tidskomplexitetet *O(n)*.

Utöver dessa inbyggda typer har CTML++ även stöd för användardefinierade klasser som datatyper. Mer om klasser går att hitta under avsnitt 2.3.9.

Vidare kan Float, Integer och Boolean omväxlas med varandra med följande specialbeteenden:

- Float till Integer: Värdet av decimaltalet avrundas nedåt till närmaste heltal.

[2]Ruby's installationsmanual
[3]Gitlab

- Integer och Float till Boolean: Då booleska värden hanteras som heltal blir 0 false (representerat som 0) och alla andra siffror true (representerat som talet i sig).
- Bool till Float och Integer: Bägge ärver det tal som representerade det booleska värdet.

Alla konstruktioner i CTML++ behöver deklareras med en en av dessa datatyper. För funktioner finns, utöver dessa, nyckelordet *void* som säger att funktionens returvärde inte kan vara av någon datatyp (tomt eller inget returvärde).

2.3.2 Operatorer

Nedan visas en tabell på alla operatorer som existerar i CTML++ tillsammans med kodexempel på hur dessa kan användas.

Tabell 1: Operatorer

Operator	**Beskrivning**	**Exempel**
+ -	Addition och subtraktion	<1+2/>/ → 3
* /	Multiplikation och division	<8/2/> → 4
%	Modulo	<5%4/> → 1
^	Potens	<2^3/> → 8
== != < > >= <=	Jämförelse	<5<2/> → false
&& ||	AND och OR	<false && true/> → false
!	NOT	<! false/> → true
- +	Unärt minus och plus	<--1/> → 1
=	Tilldelning	<a=5/>

2.3.3 Print

Print är en inbyggd funktion som kan användas för att skriva ut värden i terminalen vid körning.

```
<int heltal = 5/>
<print(heltal)/>
<print(true)/>
```

Listing 1: Exempel på print-funktion.

Detta skriver ut värdena 5 och 1 i terminalen.

2.3.4 Variabeltilldelning

I CTML++ kan du tilldela variabler värden. Vid deklaration av en variabel måste en datatyp anges, vilket binder variabeln med datatypen.

```
<int heltal = 2/>
<float decimaltal = 2.0/>
<bool trueOrFalse = true/>
```

Listing 2: Exempel på variabeldeklaration och tilldelning.

En variabel som har blivit tilldelad ett värde kan ändra värde senare i programmet. Vid omtilldelning behöver en datatyp inte anges igen, men det nya värdet måste vara av samma typ som variabeln ursprungligen var bunden till. Undantag för detta är omväxling mellan de inbyggda typerna Boolean, Integer och Float, som kan tilldelas till varandra enligt de särskilda konverteringsreglerna beskrivna under avsnittet 2.3.1.

```
<heltal = 3/>
<decimaltal = 3.0/>
<trueOrFalse = false/>
```

Listing 3: Exempel på omtilldelning av variabler (fortsättning av listing 2).

2.3.5 If-sats

En villkorssats, även kallad if-sats, används för att styra programmet baserat på om ett villkor är sant eller falskt. Kodblocket inom en if-sats körs endast om villkoret är uppfyllt, annars ignoreras det.

```
<if 5 > 2/> -> true
    <print(true)/>
</if>
<if 5 < 2/> -> false
    <print(false)/>
</if>
```

Listing 4: Exempel på if-sats.

I exemplet ovan kommer endast det första kodblocket att köras, eftersom 5>2 är sant. Detta resulterar i att 1 skrivs ut i terminalen då true hanteras i form av heltal. Det andra kodblocket körs inte, eftersom villkoret 5<2 är falskt.

2.3.6 While-loop

En while-loop upprepar ett kodblock så länge det angivna villkoret är sant.

```
<int heltal = 0/>
<while heltal < 3/>
    <print(heltal)/>
    <heltal = heltal + 1/>
</while>
```

Listing 5: Exempel på while-loop.

I exemplet ovan kommer kodblocket att köras så länge variabeln **heltal** är mindre än 3. För varje varv skrivs det aktuella värdet av **heltal** ut i terminalen, och sedan ökas värdet med 1. Detta resulterar i följande utskrift: 0, 1, 2. Villkoret utvärderas innan varje iteration enligt följande:

1. 0<3 → sant
2. 1<3 → sant
3. 2<3 → sant
4. 3<3 → falskt

När villkoret blir falskt avslutas loopen och kodblocket körs inte längre.

2.3.7 Funktioner

I CTML++ utgör funktioner kodblock som till viss del är oberoende av resten av programkoden. De tar inte hänsyn till vad som händer utanför sig själv, om de inte använder variabler som är deklarerade globalt. I första hand används den kod som finns inom funktionsblocket eller de parametrar som skickats in. Syftet är främst att skapa återanvändbara kodavsnitt och därmed motverkar koduppprepning. Varje funktion måste deklareras med en datatyp före funktionsnamnet, vilket innebär att den måste returnera ett värde av samma typ. Om void används som returtyp får inget värde returneras, om användaren skriver en return utan värde returneras automatiskt 0. Det är endast tillåtet att deklarera funktioner i det yttersta blocket eller inom klassdefinitioner, alltså går det exempelvis inte att definiera en funktion i en if-sats.

```
<int foo(int a, int b)/>
    <ret a/>
</foo>

<void foo2()/>
    <print(10)/>
    <ret/>
</foo2>

<print(foo(1, 2))/>
<print(foo2())/>
```

Listing 6: Exempel på funktioner.

Koden i exemplet ovan kommer skriva ut 1, 10 och 0 i terminalen. 1 skrivs ut då funktionen *foo(int a, int b)* returnerar **a**, och i anropet sätter vi **a** till 1. Funktionen anropas inom en print-funktion vilket betyder att det som returneras skrivs ut. 10 skrivs ut när funktionen *foo2()* körs, då det ligger en print med värde 10 i kodblocket. 0 skrivs ut på grund av att anropet till funktionen ligger i en print-funktion.

2.3.8 Behållare, LList

I CTML++ finns **LList** som behållare. LList är implementerad som en länkad lista med stöd för *random access*, samt insättning och borttagning av element längst fram i listan. Till skillnad från behållare i C++ är elementen i en LList inte starkt typade, vilket innebär att de kan innehålla värden av olika datatyper utan någon typkontroll.

Funktioner tillhörande LList är:

- *LList*.pop_front() - Tar bort elementet längst fram. *O(1)* tidskomplexitet.
- *LList*.push_front(*value*) - Lägger till ett element *value* längst fram. *O(1)* tidskomplexitet.
- *LList*[*index*] - Hämtar värdet av ett element på plats *index*. *O(n)* tidskomplexitet.

```
<llist lista = {1, {2.0, {3}}, true}/>
<bool a = lista[3]/> -> a blir true
<lista.pop_front()/> -> list blir {{2.0, {3}}, true}
<lista.push_front(5)/> -> list blir {5, {2.0, {3}}, true}
```

Listing 7: Exempel på listor och dess funktioner.

2.3.9 Klasser

CTML++ har stöd för användardefinierade klasser vilket gör det möjligt att skapa egna objekt med inre medlemsvariabler och metoder. Klasser måste definieras i det globala scopet, och har sina egna inre scope där metoder och medlemsvariabler måste definieras i ett **private** eller **public** block för att sätta dess åtkomstmodifierare. Konstruktioner inom ett publikt block kan nås utanför klassdefinitionen medan de i det privata blocket endast nås innanför.

En klass kan även ärva publika konstruktioner från en annan klass. Detta görs vid klassdefinitionen genom att ange namnet på den andra klassen man vill ärva från tillsammans med åtkomstmodifierarna public eller private. Vid arv med public kommer alla konstruktioner som ärvs ha samma åtkomsmodifierare som de initialt fanns under. Vid arv med private kommer istället alla konstruktioner ha åtkomstmodifieraren private vilket innebär att de inte kommer kunna kallas på utanför klassen, även om de initialt låg under public.

Nedan kommer ett förtydligande exempel på allt som beskrivs ovan och hur klasser kan användas:

```
<class Animal/>
    <private/>
        <int legs = 0/>
        <bool yes()/>
            <ret true/>
        </yes>
    </private>
    <public/>
        <bool isAnimal()/>
            <ret yes()/>
        </isAnimal>
    </public>
</Animal>

<class Dog : public Animal/>
    <public/>
    <void setLegs(int x)/>
            <legs = x/>
    </setLegs>
    <int getLegs()/>
            <ret x/>
    </getLegs>
    </public>
</Dog>

<Dog chihuahua/> -> Skapar en instans av klassen Dog.

<print(chihuahua.isAnimal())/> -> true

<chihuahua.setLegs(4)/>
<print(chihuahuha.getLegs())/> -> 4

<print(chihuahuha.yes())/> -> ERROR: Funktionen yes ligger under private.
```

Listing 8: Exempel på klasser och arv.

2.3.10 Polymorfism

I CTML++ kan subklasser definieras och hanteras som instanser av deras superklasser, och därav även skriva över metoder som definierats av sina superklasser. Detta möjliggör ett polymorfiskt beteende där superklasser kan fungera som ett gemensamt gränssnitt för subklasser, och där superklassens metoder kan skrivas över vid behov.

För definiera en klassmetod som kan skrivas över används taggen ***<virtual/>*** innan metoddefinitionen, och för att sedan skriva över den metoden i en subklass används taggen ***<override/>*** innan metoddefinitionen. Viktigt är att den överskrivande metodens namn och parametrar matchar originalmetoden.

Nedan visas ett exempel på hur polymorfiska klasser kan definieras och användas:

```
<class Animal/>
    <public/>
    <virtual/> <int makeSound()/><ret sound()/></makeSound>
    <virtual/> <int sound()/><ret 111/></sound>
    <int baseOnly()/><ret 999/></baseOnly>
    </public>
</Animal>

<class Dog : public Animal/>
    <public/>
    <override/> <int sound()/><ret 222/></sound>
    </public>
</Dog>

<class GuardDog : public Dog/>
    <public/>
    <override/> <int sound()/><ret 333/></sound>
    </public>
</GuardDog>

<class Cat : public Animal/>
    <public/>
    <override/> <int sound()/><ret 444/></sound>
    </public>
</Cat>

<Animal a1 = Animal/>
<Animal a2 = Dog/> -> skapa en instans av Dog som hanteras som Animal
<Animal a3 = GuardDog/>
<Animal a4 = Cat/>

<print(a1.makeSound())/> -> 111
<print(a2.makeSound())/> -> 222
<print(a3.makeSound())/> -> 333
<print(a4.makeSound())/> -> 444

<print(a1.baseOnly())/> -> 999
<print(a2.baseOnly())/> -> 999
<print(a3.baseOnly())/> -> 999
<print(a4.baseOnly())/> -> 999
```

Listing 9: Exempel på klasser och polymorfism.

3 Systemdokumentation

Detta avsnitt beskriver hur CTML++ är uppbyggt internt. Från hur en fil läses in och tokeniseras, till uppbyggnaden av syntaxträdet, evaluering av noder samt hantering av scopes.

3.1 Start

För använda CTML++ kan du köra det i interaktivt läge genom att köra filen **parser.rb** med kommandot: *ruby parser.rb*. I interaktivt läge kan du skriva kod direkt i terminalen.

För att köra en fil använder du kommandot: *ruby parser.rb <filnamn>*. Filen kan ligga i samma katalog som **parser.rb** eller i en underkatalog. **parser.rb** kommer att söka efter en fil med det angivna namnet. Om filen hittas körs den, annars skrivs ett felmeddelande ut i terminalen.

3.2 Tokens

När filen har lästs in matchas innehållet mot ett antal fördefinierade tokens. Detta görs med hjälp av parsern `rdparse`, som vi har fått tillgång till i kursen. I CTML++ har vi valt att spara tokens för nyckelord såsom **if**, olika typer av tal såsom heltal och decimaltal, samt variabelnamn. Matchningen för variabelnamn säger att användaren kan ange valfri sträng, men strängen måste börja med ett understreck eller en bokstav.

```
token(/if/) {:if} -> nyckelordet if
token(/\d+/) {|m| m.to_i } -> heltal
token(/\d+\.\d*/) {|m| m.to_f } -> decimaltal som 1.
token(/\d*\.\d+/) {|m| m.to_f } -> decimaltal som .1
token(/_*[a-zA-Z]+\w*/) {|m| m} -> variabelnamn
```

Listing 10: Exempel på tokenisering.

När innehållet har blivit tokens matchas dessa med de regler vi har definerat, se 4 för en fullständig beskrivning av regler. Om en regel matchar till fullo skapas en nod som representerar den aktuella matchningen. Detta upprepas tills samtliga tokens har matchat en regel och genererat motsvarande noder.

3.3 Noder

Grundstrukturen för en nod i CTML++ består av två funktioner, *def initialize()* som körs när noden skapas och sätter värden på alla medlemsvariabler utifrån givna inparameter, och *def eval(scope)*, som körs när noden evalueras. I alla noders eval-funktion skickas **scope** med som parameter. Detta är en del av vår scopehantering och gör att noden evalueras utifrån det scope den befinner sig i.

Enstaka noder har ytterligare funktioner utöver de två tidigare nämnda. Anledningen till detta är att de kan påverkas av att en annan nod evalueras. Exempelvis, när en nod som representerar en inparameter till en funktion evalueras, måste ett värde tilldelas utan att noden själv behöver evalueras.

3.4 Syntaxträd

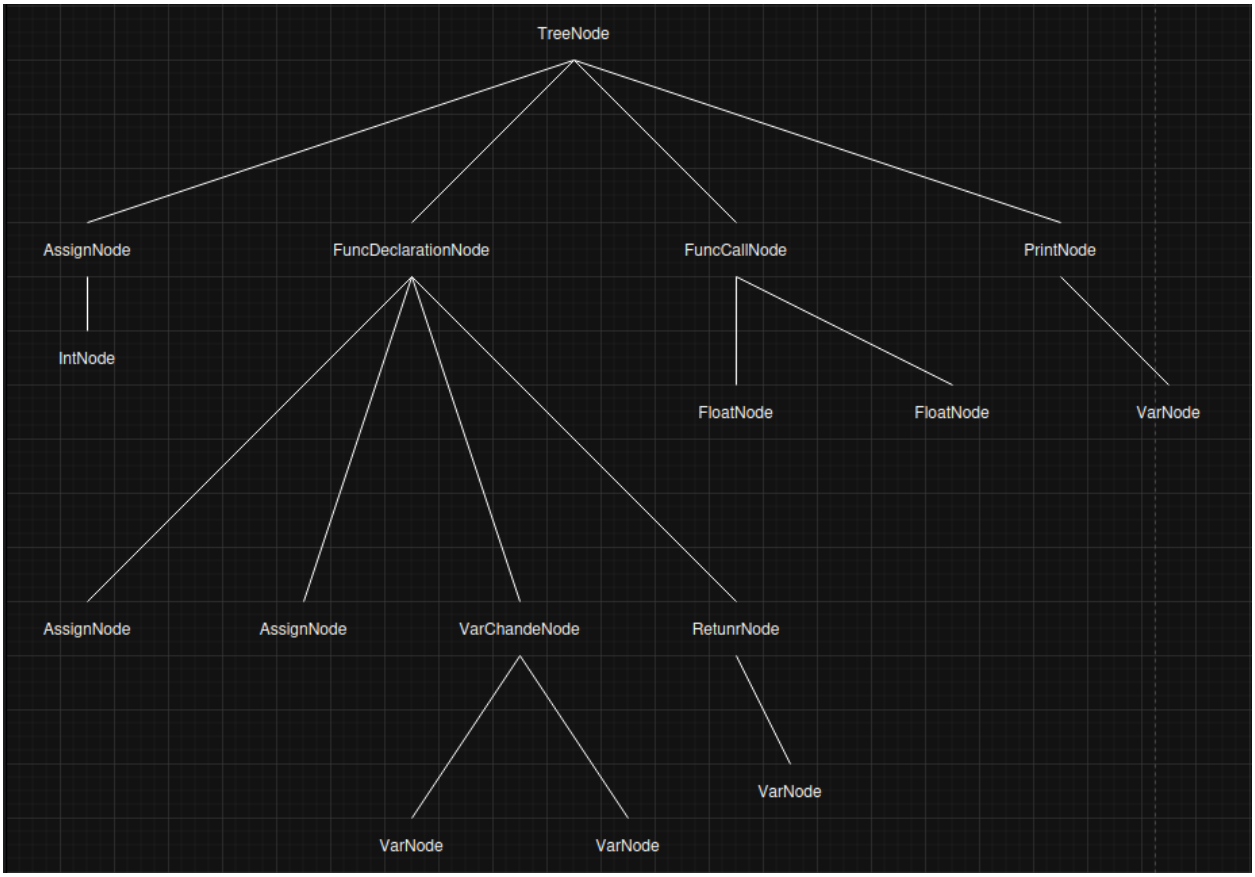
Noderna bygger upp ett syntaxträd i den ordning dem skapas i. Det vill säga att en nod som skapas tidigt hamnar längre upp i syntaxträdet, och om noderna hänger ihop fylls det på med noder under den första noden. Då syntaxträdet evalueras nedifrån och upp så kan vi med denna struktur skapa korrekt prioritet baserat på ordningen vi skapar noderna. Exempelvis, matcha en additions-nod tidigare än en multiplikations-nod.

```
<int a = 0/>
<int foo(int x, int y)/>
    <a = x/>
    <ret a/>
</foo>

<foo(5.0, 5.0)/>
<print(a)/>
```

Listing 11: Exempel för att se hur syntaxträd byggs upp.

Koden ovan skapar syntaxträdet i figur 1, se avsnitt 3.4.



Figur 1: Syntaxträd

3.5 Evaluering

Evaluering sker rekursivt från toppen till botten av syntaxträdet. Varje nod anropar eval-funktionen för sina underliggande noder. När en nod utan underliggande noder evalueras, returneras dess värde rekursivt uppåt i trädet.

3.6 Scope

Scopet i CTML++ är i grunden en hash med fyra olika nycklar:

1. **parent** - har som värde en ny hash. **Parent** används för att spara det överliggande scopet. Exempelvis sparar en if-sats det scope som ligger direkt ovanför, medan en funktion alltid har det globala scopet som parent.
2. **stmts** - står för statementsöch har som värde en lista. I listan sparas alla noder som ska evalueras.
3. **vars** - har som värde en ny hash. I denna hash sparas alla assign-noder och funktionsdeklarations-noder.
4. **userClasses** - har som värde en ny hash. I denna hash sparas alla class-noder. Denna finns endast i det globala scopet.

Denna hash skickas med till alla noders eval-funktion, så att noden kan evaluera korrekt utifrån vilket scope den ligger i. Detta gör också att när ett tidigare scope når exempelvis en if-nod, kan noden skapa ett nytt scope där den sätter scopet som skickades som parameter till parent.

Klasser har sina egna scope med utöver det ovan även nycklarna:

1. **private** - innehåller alla konstrutkioner definierade under den privata åtkomstmodifieraren. Alla konstruktioner med publik åtkomstmodifierare läggs i *stmts*.
2. **vtable** - *virtual function table* som har koll på alla virtuella metoder och vilken specifik implementation de representeras av baserad på om de skrivs över eller inte i en subklass.
3. **name** - namnet på klassen.

3.7 Runtime.rb

I projektfilerna tillhörande CTML++ finns en fil kallad **runtime.rb**, som innehåller hjälpfunktioner som används av flera olika noder. Syftet med denna fil är att motverka kodupprepning. En funktion som många noder använder är findscope-funktionen, som söker efter variabler som deklarerats i tidigare scope.

```
def findScope(scope, name)
    currentScope = scope
    while currentScope["parent"] != nil
        if currentScope["vars"][name]
            return currentScope
        else
          currentScope = currentScope["parent"]
        end
      end
    if currentScope["parent"] == nil
        if currentScope["vars"][name]
            return currentScope
        end
    end
    raise Error::NameError, "Variable #{name} hasn't been declared yet"
end
```

Listing 12: findscope-funktion.

3.8 Felhantering

Felhantering är implementerad för situationer där parsern lyckas tokenisera koden, men användaren försöker utföra operationer som inte är tillåtna i CTML++. Ett exempel på detta är när användaren försöker använda en variabel som antingen inte är deklarerad eller inte är tillgänglig i det aktuella scopet. I sådana fall kastas ett fel med ett meddelande som förklarar vad som gått fel. För kodexempel, se 3.7.

4 Grammatik

Backus-Naur form

⟨*start*⟩ ::= ⟨*start*⟩ ⟨*classdec*⟩
| ⟨*classdec*⟩

⟨*classdec*⟩ ::= '<' 'class' ⟨*varname*⟩ ⟨*heritage*⟩ '/>' ⟨*priv*⟩ ⟨*pub*⟩ '</' ⟨*varname*⟩ '>'
| '<' 'class' ⟨*varname*⟩ ⟨*heritage*⟩ '/>' ⟨*pub*⟩ ⟨*priv*⟩ '</' ⟨*varname*⟩ '>'
| '<' 'class' ⟨*varname*⟩ ⟨*heritage*⟩ '/>' ⟨*pub*⟩ '</' ⟨*varname*⟩ '>'
| '<' 'class' ⟨*varname*⟩ ⟨*heritage*⟩ '/>' ⟨*priv*⟩ '</' ⟨*varname*⟩ '>'

⟨*priv*⟩ ::= '<' 'private' '/>' ⟨*classblock*⟩ '</' 'private' '>'
| ''

⟨*pub*⟩ ::= '<' 'public' '/>' ⟨*classblock*⟩ '</' 'public' '>'
| ''

⟨*heritage*⟩ ::= ':' 'public' ⟨*varname*⟩
| ':' 'private' ⟨*varname*⟩
| ''

⟨*classblock*⟩ ::= ⟨*classblock*⟩ ⟨*polyfuncdec*⟩
| ⟨*polyfuncdec*⟩
| ''

⟨*funcdec*⟩ ::= '<' 'void' ⟨*varname*⟩ '(' ⟨*params*⟩ ')' '/>' ⟨*block*⟩ '</' ⟨*varname*⟩ '>'
| '<' 'void' ⟨*varname*⟩ '(' ')' '/>' ⟨*block*⟩ '</' ⟨*varname*⟩ '>'
| '<' ⟨*type*⟩ ⟨*varname*⟩ '(' ⟨*params*⟩ ')' '/>' ⟨*block*⟩ '</' ⟨*varname*⟩ '>'
| '<' ⟨*type*⟩ ⟨*varname*⟩ '(' ')' '/>' ⟨*block*⟩ '</' ⟨*varname*⟩ '>'
| ⟨*sections*⟩

⟨*classfuncdec*⟩ ::= '<' 'void' ⟨*varname*⟩ '(' ⟨*params*⟩ ')' '/>' ⟨*block*⟩ '</' ⟨*varname*⟩ '>'
| '<' 'void' ⟨*varname*⟩ '(' ')' '/>' ⟨*block*⟩ '</' ⟨*varname*⟩ '>'
| '<' ⟨*type*⟩ ⟨*varname*⟩ '(' ⟨*params*⟩ ')' '/>' ⟨*block*⟩ '</' ⟨*varname*⟩ '>'
| '<' ⟨*type*⟩ ⟨*varname*⟩ '(' ')' '/>' ⟨*block*⟩ '</' ⟨*varname*⟩ '>'
| '<' ⟨*type*⟩ ⟨*varname*⟩ '=' ⟨*boolexpr*⟩ '/>'
| '<' 'llist' ⟨*varname*⟩ '=' '{' ⟨*llistTerm*⟩ '}' '/>'

⟨*polyfuncdec*⟩ ::= '<' 'virtual' '/>' ⟨*classfuncdec*⟩
| '<' 'override' '/>' ⟨*classfuncdec*⟩
| ⟨*classfuncdec*⟩

⟨*block*⟩ ::= ⟨*block*⟩ ⟨*sections*⟩
| ⟨*sections*⟩
| ''

⟨*sections*⟩ ::= ⟨*printer*⟩
| '< 'while' ⟨*boolexpr*⟩ '/>' ⟨*block*⟩ '</' 'while' '>'
| '<' 'if' ⟨*boolexpr*⟩ '/>' ⟨*block*⟩ '</' 'if' '>'
| '<' 'llist' ⟨*varname*⟩ '=' '{' ⟨*llistTerm*⟩ '}' '/>'
| '<' 'return' ⟨*boolexpr*⟩ '/>'
| '<' 'return' '/>'
| '<' ⟨*type*⟩ ⟨*varname*⟩ '=' ⟨*boolexpr*⟩ '/>'
| '<' ⟨*type*⟩ ⟨*varname*⟩ '/>'
| '<' ⟨*vars*⟩ '=' ⟨*boolexpr*⟩ '/>'
| '<' ⟨*llistAccess*⟩ '=' ⟨*boolexpr*⟩ '/>'
| '<' ⟨*varname*⟩ '.' 'push_front' '(' ⟨*boolexpr*⟩ ')' '/>'
| '<' ⟨*varname*⟩ '.' 'pop_front' '(' ⟨*boolexpr*⟩ ')' '/>'
| '<' ⟨*varname*⟩ '.' 'pop_front' '(' ')' '/>'
| '<' ⟨*boolexpr*⟩ '/>'
| '<' ⟨*varname*⟩ ⟨*varname*⟩ '=' ⟨*varname*⟩ '/>'
| '<' ⟨*varname*⟩ ⟨*varname*⟩ '/>'

⟨*llistTerm*⟩ ::= ⟨*llistItem*⟩ ',' ⟨*llistTerm*⟩
| ⟨*llistItem*⟩

⟨*llistItem*⟩ ::= ⟨*boolexpr*⟩
| '{' ⟨*llistTerm*⟩ '}'

⟨*funcCallArgs*⟩ ::= ⟨*funcCallArgs*⟩ ',' ⟨*boolexpr*⟩
| ⟨*boolexpr*⟩

⟨*params*⟩ ::= ⟨*params*⟩ ',' ⟨*type*⟩ ⟨*varname*⟩
| ⟨*type*⟩ ⟨*varname*⟩

⟨*boolexpr*⟩ ::= '!' ⟨*boolexpr*⟩
| ⟨*andorexpr*⟩

⟨*andorexpr*⟩ ::= ⟨*diff*⟩ '&&' ⟨*boolexpr*⟩
| ⟨*diff*⟩ '||' ⟨*boolexpr*⟩
| ⟨*diff*⟩

⟨*diffops*⟩ ::= '=='
| '!='
| '<'
| '>'
| '>='
| '<='

⟨*type*⟩ ::= 'int'
| 'float'
| 'bool'
| 'llist'

⟨*varname*⟩ ::= [a-zA-Z\w]+

⟨*printer*⟩ ::= `'<' 'print' '(' ⟨boolexpr⟩ ')' '/>'`

⟨*expr*⟩ ::= ⟨*expr*⟩ `'+'` ⟨*mulDiv*⟩
| ⟨*expr*⟩ `'-'` ⟨*mulDiv*⟩
| ⟨*mulDiv*⟩

⟨*mulDiv*⟩ ::= ⟨*mulDiv*⟩ `'*'` ⟨*ex*⟩
| ⟨*expr*⟩ `'/'` ⟨*ex*⟩
| ⟨*ex*⟩

⟨*ex*⟩ ::= ⟨*term*⟩ `'^'` ⟨*ex*⟩
| ⟨*ex*⟩ `'%'` ⟨*term*⟩
| ⟨*negate*⟩

⟨*negate*⟩ ::= `'+'` ⟨*negate*⟩
| `'-'` ⟨*negate*⟩
| ⟨*term*⟩

⟨*term*⟩ ::= `'true'`
| `'false'`
| ⟨*int*⟩
| ⟨*float*⟩
| ⟨*varname*⟩ `'('` ⟨*funcCallArgs*⟩ `')'`
| ⟨*varname*⟩ `'(' ')'`
| ⟨*vars*⟩ `'.'` ⟨*varname*⟩ `'('` ⟨*funcCallArgs*⟩ `')'`
| ⟨*vars*⟩ `'.'` ⟨*varname*⟩ `'(' ')'`
| ⟨*llistAccess*⟩
| `'('` ⟨*boolexpr*⟩ `')'`
| ⟨*vars*⟩
| `''`

⟨*int*⟩ ::= d+

⟨*float*⟩ ::= d+.d*
| d*.d+

⟨*llistAccess*⟩ ::= ⟨*varname*⟩ ⟨*llistIndex*⟩

⟨*llistIndex*⟩ ::= `'['` ⟨*boolexpr*⟩ `']'` ⟨*llistIndex*⟩
| `'['` ⟨*boolexpr*⟩ `']'`

⟨*vars*⟩ ::= [a-zA-Z\w]+

5 Erfarenhet och reflektion

Projektet i helhet har varit väldigt lärorikt, roligt och utmanande. Vi har inte bara lärt oss mycket om ruby, scope-hantering och typning. Utan också mycket om bakomliggande implementationsdetaljer i C++ och andra objektorienterade språk med stark typning.

Den största utmaningen vi stötte på var vid implementeringen av klasser, främst polymorfism och objektorientering. Först var vi tvungna att lära oss om vad virtual function tables är och hur det kan används för att säkerställa att rätt metoder kallas vid polymorfism.

Sedan fastnade vi på scope-hantering för klasser, mer specifikt rekursiva funktioner inom klasser. Detta blev ett edge-case, då en klass i vanliga fall kallade på en funktion med klassens scope, medan en funktion som kallar på sig själv behöver funktionens scope. Vi löste detta genom att kolla ifall funktionsanropet var till samma funktion som det befann sig i.

Ett annat problem var typ-hanteringen hos listor. Vi hade tänkt från början att listor skulle vara som i C++ och ha stark typning, det vill säga att man inte får blanda element av olika datatyper. Men när vi väl började så upptäckte vi att det var utmanande och tidskrävande. Därför valde vi att istället lägga den tiden på att jobba med klasser.

En liten skillnad från den tänkta syntaxen var att vi lade till ett backslash innan större/mindre tecknet som stänger en konstruktion. Exempelvis så tänkte vi att en varibeltilldelning skulle vara: *<int a = 2>*, men då vi fick problem med det sista tecknet i samband med jämförelser så gjorde vi om till *<int a = 2/>*. Vidare ändrade vi syntaxen för definiering av funktioner och funktionsanrop till ett syntax som är mer likt C++ med parenteser. Exempel: funktionsanropet *<print arg = 2/>* blir *<print(2)/>*, och funktionsdefinitionen *<int func args = int a/>* blir *<int func(int a)/>* med den nya syntaxen.

Utöver detta stötte vi inte på fler problem, och vi lyckades implementera allt vi bestämt oss för i språkspecifikationen. Vi hade även en rimlig tanke om hur svårt saker skulle vara, vi tänkte att det mesta skulle flyta på om vi var noggranna med att testa så allt fungerar under hela projektets gång. Medan konstruktioner för högre betyg skulle vara mer utmanande, vilket det var.

Från detta projekt tar vi med oss vikten av att börja tidigt och planera tiden. Vi planerade från början att det skulle ta lång tid att få CTML++ så som vi vill ha det, vilket gjorde att vi började tidigt och hann med att implementera allt. Men också att man inte kan ha för mycket tester, vi gjorde misstaget när vi testade klasser, att testa en sak och sedan ta bort testerna istället för att ha kvar alla tester. Troligen så hade vi sparat mycket tid om vi sparade alla tester så vi kunde gå tillbaka och se att saker fungerade efter vi implementerat nästa sak.