LINKÖPINGS UNIVERSITET

TDP019 Projekt: Datorspråk

TDP019
Språkdokumentation Songic

Författare

Daniel Huber, danhu849@student.liu.se
Theodore Nilsson, theni230@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2021
Version 1.0

2021-06-01

Innehåll

1 Inledning

Tänk dig att du är en låtskrivare (om du inte redan är det) redo att få din senaste idé på papper. Det är inte mer än en truddelutt, men du vill förvandla den till en fullfjädrad hit för alla dina vänner att njuta av. Slutligen, efter timmar av arbete, där har du det, ditt första utkast med en färdig struktur och melodier som skulle röra vilken musikälskare som helst. Men nu, efter att ha hört låten i sin helhet, upptäcker du att det finns något som inte riktigt passar. Är det den noten i slutet av versen som inte leder lyssnaren till refrängen, eller behöver du kanske ändra strukturen helt?

I de flesta fall står du nu inför en mödosam process av att flytta runt och ändra noter en efter en tills de passar ihop precis rätt. Programmeringsspråket *Songic* försöker effektivisera denna process med ett strukturerat förhållningssätt till melodiskrivning.

Många låtskrivare vet att de kan dela upp låtar i strukturella element, som en var, en refräng eller ett stick, och att dessa i sin tur består av korta melodiska idéer som kallas motiv. Poängen med Songic är att ge användaren möjlighet att använda denna struktur för att skapa och redigera melodier för en hel låt på en gång.

Songic, ett relativt dynamiskt typat DSL för musikintresserade. Med möjlighet att underlätta utskrift av längre musikstycken genom kod. Songic är lätt att lära sig och lättare att skriva då större underliggande programmeringskunskaper ej behövs. Språket har konstruerats utifrån hur en sång/låt är uppbyggd och har funktionalitet för kontrollstrukturerna if-, for- och repetitionssatser.

1.1 Målgrupp

Songic är riktat mot den med lite programmeringserfarenhet men större musikförmåga. Songic är tänkt som ett redskap för konstruktion av musikstycken och notgenerering för den effektive som vet vilka noter ett sångstycke ska innehålla utan att skriva dem en och en för hand. Alla utskrifter av Songicprogram skrivs som en sekvens av noter.

1.2 Användarhandledning

Detta avsnitt ämnar att introducera läsaren till Songic som språk och verktyg för melodiskrivning. Efter läsning hoppas läsaren ha fått tillräckliga kunskaper för att kunna skriva egna Songicprogram. Ytterligare information om de specifika klasserna och dess begränsingar finns i Systemdokumentationen.

Ett Songicprogram utgörs av en fil med ändelsen .song och måste innehålla tre delar för att korrekt kunna exekveras: ett *motif-block*, ett *segment-block* och ett *structure-block*. Songicprogram kan skrivas på flera rader eller bara en rad. Under denna rubrik kommer en stegvis introduktion med flera exempel presenteras för att ge en fullständig bild av Songic och dess användning.

```
1 motif{}
2 segments{}
3 structure{}
```

I den första och översta strukturen, motif-blocket, måste Songics kontrollstrukturer skrivas och variabler definieras. Variabelnamn kan endast bestå av VERSALER. Främsta syftet med motifblocket är att variabler på olika sätt tilldelas noter som används i de två senare blocken. Noter känns igen direkt av parsern och behöver inte deklareras. De skrivs oftast bara med en symbol, men flera skrivsätt finns vilka kan ses nedan.

- c+3 - En fjärdedelsnot med tonhöjd C i tredje oktaven.
- 2c#+2 - En halv not med tonhöjd C höjt ett halvt steg i andra oktaven.
- d - En helnot med tonhöjd D i mellersta oktaven.

```
motifs{
A = c+3
B = 2c#+2 2c#+2
C = d
}
segments{
VERSE = A, B, C
}
structure{
VERSE
}
```

Output:

```
c+3 c#+2 c#+2 d
```

I motif-blocket finns funktionalitet för kontrollstrukturerna klassisk for-loop, en repetitionsfunktion (repeat), och if-satser. For-loopen utgörs av nyckelordet for, ett matematiskt uttryck och ett block av kod inom hakparenteser. Antalet iterationer specificeras av det matematiska uttrycket som kan evalueras till ett heltal. Kod inom for-loopens hakparenteser exekveras i ett högre scope än kod utanför hakparenteserna. Därför försakas variabeln B medan variabeln A ges en annan notsekvens i koden nedan.

```
motifs{
A = a b c
for 3 [
A = g
]
for 2 plus 2 [ B = a b c ]
}
segments{
VERSE = A
}
structure{
VERSE
}
```

Output:

```
g
```

Av funktionen repeat returneras antalet sekvenser av de specificerade noterna innanför repeats hakparenteser. De båda variablerna nedan tilldelas samma notsekvens.

```
motifs{
G = repeat 3 [ g ]
H = g g g
}
segments{VERSE = G, H}
structure{VERSE}
```

Output:

```
g g g g g g
```

Kontrollstrukturen if kan skrivas på två olika sätt. I första sättet tilldelas variabeln noterna inom if-satsens hakparenteser om jämförelsen är sann.

```
motifs{
D = if 3 equals 3 [d d]
A = if true or false [a g]
E = if true and true [a b]
}
segments{VERSE = D, A}
structure{VERSE}
```

Output:

```
d d a g
```

Det andra sättet är en fristående if-sats också med en jämförelse samt hakparenteser innehållandes variabeltilldelningar.

```
motifs{
A = a b c
if 3 equals 3 [ A = b ]
if true or false [
A = a
]
E = if true and true [B = b]
}
segments{VERSE = A}
structure{VERSE}
```

Output:

```
a
```

I det andra blocket segments kombineras olika motifs ihop till de strukturella beståndsdelarna av en sång, såsom verser och refränger. Det rekommenderas att segmenten i detta block tilldelas passande namn även om inget sådant inbyggt krav finns. Ett segment skapas genom att en variabel tilldelas flera kommaseparerade motif.

```
motifs{
A = 2a 2b 2c
B = d+2 e# 2a+2
C = 2b 8z
}
segments{
VERSE = A, B, C, C
CHORUS = B, B, B, A
}
structure{
VERSE, CHORUS
}
```

Output

```
2a 2b 2c d+2 f 2a+2 2b 8z 2b 8z d+2 f 2a+2 d+2 f 2a+2 d+2 f 2a+2 2a 2b 2c
```

Inuti det sista blocket, structure, läggs de olika segmenten ihop i den ordningen de är menade att spelas. Returen från programmet resulterar i en utskrift av noter i den ordningen.

```
motifs{
A = 2a 2b 2c
B = d+2 e# 2a+2
C = 2b 8z
}
segments{
VERSE = A, B, C, C
CHORUS = B, B, B, A
}
structure{
VERSE, CHORUS, VERSE, CHORUS, CHORUS
}
```

Output:

```
2a 2b 2c d+2 f 2a+2 2b 8z 2b 8z d+2 f 2a+2 d+2 f 2a+2 d+2 f 2a+2 2a 2b 2c 2a 2b 2c d+2 f 2a+2 2b
    8z 2b 8z d+2 f 2a+2 d+2 f 2a+2 d+2 f 2a+2 2a 2b 2c d+2 f 2a+2 d+2 f 2a+2 d+2 f 2a+2 2a 2b 2c
```

2 Systemdokumentation

Språket Songic är konstruerat utifrån parsern rdparse och Diceroller exemplet tillgänglig från kursen TDP007. Inspiration har tagits utifrån tankesätten om att ha multipla ihopsittande noder som bygger upp ett programs exekveringssekvens. Dessa noder placeras under parsning i en rot-not, kallad *@@root_node*. När parsningen avslutats evalueras hela rotnoden, koden exekveras och returen skrivs ut.

2.1 BNF Grammatik

BNF ::::::::::

⟨*song*⟩ ::= ⟨*motif_block*⟩ ⟨*segment_block*⟩ ⟨*structure_block*⟩

⟨*structure_block*⟩ ::= structure ⟨*segments*⟩

⟨*segments*⟩ ::= ⟨*segments*⟩ ‘`,`’ ⟨*var*⟩
| ⟨*var*⟩

⟨*segment_block*⟩ ::= ‘`segments`’ ⟨*segment_variable_assignments*⟩

⟨*segment_variable_assignments*⟩ ::= ⟨*segment_variable_assignments*⟩ ⟨*segment_variable_assignment*⟩
| ⟨*segment_variable_assignment*⟩

⟨*segment_variable_assignment*⟩ ::= ⟨*segment_variable_assignment*⟩ ‘`,`’ ⟨*var*⟩
| ⟨*var*⟩ = ⟨*var*⟩
| ⟨*var*⟩ = ⟨*loop*⟩

⟨*motif_block*⟩ ::= ‘`motifs`’ ⟨*motif_statements*⟩

⟨*motif_statements*⟩ ::= ⟨*motif_statements*⟩ ⟨*motif_statement*⟩
| ⟨*motif_statement*⟩

⟨*motif_statement*⟩ ::= ⟨*var*⟩ ‘`=`’ ⟨*if*⟩
| ⟨*var*⟩ ‘`=`’ ⟨*motif*⟩
| ⟨*var*⟩ ‘`=`’ ⟨*loop*⟩
| ⟨*var*⟩ ‘`=`’ ⟨*expression*⟩
| ⟨*if*⟩
| ⟨*for_loop*⟩
| ⟨*loop*⟩

⟨*if*⟩ ::= ‘`IfToken`’ ⟨*expression*⟩ ⟨*comparator*⟩ ⟨*expression*⟩ ‘`[`’ ⟨*statements*⟩ ‘`]`’
| ‘`IfToken`’ ⟨*boolean*⟩ ⟨*comparator*⟩ ⟨*boolean*⟩ ‘`[`’ ⟨*statements*⟩ ‘`]`’

⟨*comparator*⟩ ::= ‘`EqualsToken`’
| ‘`OrToken`’
| ‘`AndToken`’
| ‘`LesserToken`’
| ‘`GreaterToken`’

⟨*boolean*⟩ ::= ‘`TrueToken`’
| ‘`FalseToken`’

⟨*for_loop*⟩ ::= ‘`ForToken`’ ⟨*expression*⟩ ‘`[`’ ⟨*statements*⟩ ‘`]`’

⟨*loop*⟩ ::= ‘`RepeatToken`’ ⟨*expression*⟩ ‘`[`’ ⟨*statements*⟩ ‘`]`’
| ‘`RepeatToken`’ ⟨*var*⟩ ‘`[`’ ⟨*statements*⟩ ‘`]`’

⟨*statements*⟩ ::= ⟨*statements*⟩ ⟨*statement*⟩
| ⟨*statement*⟩

⟨*statement*⟩ ::= ⟨*var*⟩ '`=`' ⟨*motif*⟩
| ⟨*var*⟩ '`=`' ⟨*expression*⟩
| ⟨*motif*⟩

⟨*motif*⟩ ::= ⟨*notes*⟩
| ⟨*note*⟩

⟨*notes*⟩ ::= ⟨*notes*⟩ ⟨*note*⟩
| ⟨*note*⟩ ⟨*note*⟩

⟨*note*⟩ ::= ⟨*note*⟩ '`.`' ⟨*method*⟩ '`(`' ⟨*expression*⟩ '`)`'
| ⟨*length*⟩ ⟨*tone*⟩ ⟨*octave*⟩
| ⟨*length*⟩ ⟨*tone*⟩
| ⟨*tone*⟩ ⟨*octave*⟩
| ⟨*tone*⟩
| ⟨*silence*⟩

⟨*method*⟩ ::= '`transposed`'

⟨*expression*⟩ ::= ⟨*expression*⟩ '`AdditionToken`' ⟨*term*⟩
| ⟨*expression*⟩ '`SubtractionToken`' ⟨*term*⟩
| ⟨*term*⟩ '`AdditionToken`' ⟨*term*⟩
| ⟨*term*⟩ '`SubtractionToken`' ⟨*term*⟩
| ⟨*term*⟩

⟨*term*⟩ ::= ⟨*factor*⟩ '`MultiplicationToken`' ⟨*factor*⟩
| ⟨*factor*⟩ '`DivisionToken`' ⟨*factor*⟩
| ⟨*factor*⟩

⟨*factor*⟩ ::= '`(`' ⟨*expression*⟩ '`)`'
| ⟨*var*⟩
| '`Integer`'

⟨*silence*⟩ ::= ⟨*length*⟩ /[z]/
| /[z]/

⟨*length*⟩ ::= '`Integer`'

⟨*octave*⟩ ::= /[+][0-9]/
| /[-][0-9]/

⟨*tone*⟩ ::= /[a-g][#|b]?/

⟨*var*⟩ ::= /[A-Z]/

2.2 Lexikalisk analys & Tokens

Lexern rdparse har använts och dess tokeniseringssystem har byggts ut för att anpassas till Songic språket. I början kastas alla mellanrum och nyradstecken. Då gemener reserverats för notigenkänning tokeniseras först nyckelorden för artimetiken, funktionerna, värdena för sant och falskt samt jämförelseoperatorerna. Nyckelorden känns igen med hjälp av regexuttryck och utifrån texten skapas antingen tomma klasser eller placeholder klasser vars innehåll är en enkel strängrepresentation. Exempelvis är jämförelseoperatorns `and`-token innehåll strängen `''and''`. Tecken utöver nyckelord returneras var för sig antingen som strängar eller som heltal.

2.3 Parsning

Ett RootNode objekt, `@@root_node`, skapas i samband med parsningen. Exekveringsordningen representeras av `@@root_node` som under parsningen fylls på med objekt vartefter de matchas. Objekten kan antingen returnera noter till andra objekt eller uppfylla tidsspecifika funktioner såsom variabeltilldelningar och variabelkontroll för att ta reda på vilket för tiden nuvarande värde en variabel har. När alla objekt lagts till från parsningen aktiveras `@@root_node`s .seval funktion.

2.4 Evaluering

`@@root_node`s .seval funktion itererar över varje objekt som finns inuti och kallar på dess egen .seval funktion. Antingen påverkas stackobjektet `$stack` vars funktion är att hantera minnet under runtime eller så byggs strängen för notsekvensen på som sedan returneras när alla objekt itererats över.

2.5 Klasser och dess relationer

2.5.1 Stack

Programmets minne och scopehantering sköts av klassen `Stack`. `Stack` ärver `Array` och ges vid sin skapelse en `Hash`. Med klassfunktionen `push_frame` adderas en ny hash till stacken och därmed en ny nivå till scopet. Med klassfunktionen `pop_frame` uppdateras alla gemensamma variabler mellan det övera och undre scopet innan det övre scopet tas bort. Av klassfunktionen `look_up(name)` returneras objektet med nyckeln name om det finns, om det inte finns så returneras `false`. Objekt adderas till `Stack` med `add`.

2.5.2 RootNode

Exekveringsordning i ett Songic program sköts av klassen `RootNode`. Klassen `RootNode` ärver `Array` och under parsningen byggs exekveringsordningen upp. Inuti klassen finns funktionen `seval` som aktiveras i slutet på parsningen vars funktion är att gå igenom varje objekt inuti `RootNode`. Då resultatet av varje Songicprogram är att generera en sekvens noter skapas först en tom sträng. Är objektets klass en sträng kollas det upp vilket objekt innehållandes noter i stacken strängen pekar på och noterna läggs till i strängen. Är objektet inte en sträng utförs objektets egenspecifika funktion med `.seval`.

2.5.3 LookUpNode

Hjälpklass vars funktion är att returnera det vid tiden aktuella värdet på variabeln `@var_name` med hjälp av `Stack`s funktion `look_up`.

2.5.4 IntegerNode

Placeholder klass för heltal ej associerade med noter. Används i aritmetik, jämförelser och som funktionsparameter.

2.5.5 StringNode

Placeholder klass för strängar. Används främst för betydelseöverföring från tokens.

2.5.6 Note

En ton kan enklast beskrivas som en konstruktion bestående av två delar, dess höjd och dess längd. Klassen *Note* reflekterar detta med medlemsvariablerna *@length* och *@halfstep*.

@length är av datatypen *Rational* och initieras med ett heltal som representerar hur många n:te-delar av en helton *Note*:n ska pågå i. Detta uppnås genom sätta heltalet som nämnare till *@length*. Eftersom det kan

argumenteras att 1/4-dels notlängd är den vanligaste notlängden har denna satts som standardvärde. För att representera en helnot ska användaren därför skriva en '1':a framför, exempelvis 1a.

@halfstep är en heltalsrepresentation av en absolut tonhöjd. Det är lättast att beskriva detta som tangenter på ett piano där varje tangent har ett eget heltal som bestäms av hur många steg från en bestämd tangent den befinner sig. Då bestäms heltalets tecken av tangentens placering till höger eller vänster på pianot. Då har lägre toner ett negativt heltal och högre toner har ett positivt heltal. Dessa heltal utgår enligt praxis av mellersta C som då får heltalet 0.

Detta heltal kan beräknas med medlemsfunktionerna *octave_to_halstep* och *tone_to_halfstep*.

octave_to_halfstep tar det givna heltalet för vilken oktav en ton ska vara i och multiplicerar den med 12 för att ge heltalet för c i den oktaven. Detta fungerar eftersom det finns 12 tangenter, eller halvtoner, i varje oktav.

tone_to_halfstep tar de givna symbolerna för noten och översätter dessa till det heltal som representerar noten i den 0:te oktaven, det vill säga den vi utgår ifrån. Eftersom namngivningen av toner inte är reguljärt över detta intervall var det enklast att helt enkelt göra detta med hjälp av en hash där varje tänkbar inmatning används som nyckel för att hitta ett heltal. Här kan även icke-korrekta inmatningar (så som e, som egentligen ska skrivas som f) tolkas till den ton som användaren med störst sannolikhet menat.

Vid evaluering returneras en stängrepresentation av noten genom att i princip göra en omvänd beräkning. Motiveringen till att representera noterna på detta sätt internt är att det lättare ska gå att göra modifikationer på de. Den inbyggda funktionen *transposed* hade exempelvis blivit mycket svårare att implementera om man var tvungen att ta hänsyn till hur dur-skalan var uppbyggd.

Det finns också en medlemsvariabel som heter *@scale*. Den är en medlemsvariabel därför att vi hoppats kunna implementera ett sätt att hantera skalor på ett smart sätt. Den finns kvar därför att den används som ett sätt att smidigt skapa strängrepresentationer för noter. Tanken var att man skulle kunna ändra skalan, men använda samma funktioner för att skriva ut noten och på så sätt hantera utskriften av olika skalor.

2.5.7 Silence

Eftersom en paus i en melodi inte kan representeras av en tangent på ett piano, eller finns i någon särskild oktav, så skiljer den sig från en not märkvärt. Därför finns det ett särskilt objekt för denna. Pausen representeras av tecknet 'z' och längden av den representeras på samma sätt som en ton, men ett haltal som motsvarar nämnaren i en kvot.

2.5.8 VariableAssignmentNode

`VariableAssignmentNodes` används som placeholders för när variabeltilldelningar ska läggas till eller uppdateras i stacken. De läggs till i `@@root_node` under parsning och evalueras under runtime.

2.5.9 Motif

Motif är en klass som agerar behållare för en sekvens med noter. Dess uppgift är att säkerställa att ett motif endast består av noter (eller pauser), och att säga till noterna den håller i att det är dags att evalueras. Märk att ett motif inte kan adderas till ett annat motif.

2.5.10 Segment

Segment-klassen har samma funktion och syfte för motif som *Motif*-klassen har för noter. Syftet med detta är att närmare efterlikna praxis för låtstruktur.

2.5.11 AddNode

Används vid runtime för att lägga till"en sak till en annan. *AddNode* ges vid parsning nycklar till ursprungsobjektet, och det som ska adderas till ursprungsobjektet. Vid runtime används dessa nycklar för att hitta objekten i stacken.

2.5.12 MathNode

Placeholderklass för typ av samtliga matteoperatorer för att underlätta jämförelsen i klassen `Repeat`.

2.5.13 Addition

Med `.seval` returneras en ruby `Integer` av additionen mellan två `IntegerNode`r.

2.5.14 Subtraction

Med `.seval` returneras en ruby `Integer` av subtraktionen mellan två `IntegerNode`r.

2.5.15 Multiplication

Med `.seval` returneras en ruby `Integer` av multiplikationen mellan två `IntegerNode`r.

2.5.16 Division

Med `.seval` returneras en ruby `Integer` av divisionen mellan två `IntegerNode`r.

2.5.17 ComparatorNode

Hjälpklass för klassen `IfNode` där booleska jämförelser evalueras och returneras. Initialiseras med ett vänsterled, ett högerled och en jämförelseoperator. Följande jämförelseoperatorer kan hanteras:

- VL `equals` HL - Ett ruby `true` returneras om både vänsterled (VL) och högerled (HL) är lika. Annars returneras ett ruby `false`.
- VL `and` HL - Ett ruby `true` returneras om både VL och HL är sanna. Annars returneras ett ruby `false`.
- VL `or` HL - Ett ruby `true` returneras om något av VL eller HL är sant. Annars returneras ett ruby `false`.
- VL `is less than` HL - Ett ruby `true` returneras VL är mindre än HL. Annars returneras ett ruby `false`.
- VL `is more than` HL - Ett ruby `true` returneras VL är större än HL. Annars returneras ett ruby `false`.

ComparatorNode begränsas av att inte jämförelser mellan heltal och booleska värden tillåts. Booleska värden representeras av antingen `true` eller `false` och inte av siffrorna ett och noll som är fallet för andra programmeringsspråk.

2.5.18 BooleanNode

Placeholder klass för en ruby boolean. Skapas med strängrepresentationen av en `TrueToken` eller `FalseToken` som inparameter. Kan evalueras till att antingen vara sann eller falsk.

2.5.19 Repeat

Tilldelningsfunktion där en uppsättning noter returneras `@iterations` gånger och kan inte skrivas som en fristående funktion. `@iterations` kan antingen vara en `IntegerNode`:

```
A = repeat 10 [ a b c]
```

eller en `IntegerNode` som representeras av en variabel:

```
B = 2
A = repeat B [ a b c]
```

2.5.20 IfNode

Klassisk if-sats med jämförelse och uppgiftslista. `IfNode`-objektet läggs till i `RootNode` om dess jämförelsen är sann. Jämförelsen tas omhand av `ComparatorNode` och kan bestå av två aritmetiska uttryck eller två booleska värden.

I Songic kan if-satser skrivas på två olika sätt. Antingen som fristående funktion:

```
B = a
if 6 is more than 5 [ B = f f f ]
```

eller som tilldelningsfunktion:

```
A = if true equals true [ a b c ]
```

Kod inuti if-satsens hakparenteser befinner sig i ett högre scope så variabler som ej finns sedan innan som tilldelas värden kommer ej att finnas när if-satsen exekverats. Exempelvis kommer följande program att krascha då B inte kommer finnas.

```
motifs{
if 3 equals 3 [ B = b c d ]
}
segments{VERSE = B}
structure{VERSE}
```

2.5.21 ForNode

Enkel for-sats där antal iterationer av en exekveringslista bestäms av ett tal eller ett matematiskt uttryck. `ForNode`ns exekveringslista utgörs av de uppgifter inuti dess hakparenteser. Uppgifterna inom hakparenteserna befinner sig ett scope

```
for 3 [ A = a b c ]
```

Liksom för if-satsen så exekveras kod innanför hakparenteserna i ett högre scope.

2.6 Kodstandard

Ingen specifik kodstandard har valts då språket designats efter att kunna skrivas av personer utan programmeringskunskaper. Även om kontrollstrukturer och aritmetik finns kan Songics huvudsakliga uppgift uppfyllas utan att dessa nyttjas. Då gemener används i parsningen av noter har beslutet tagits att variabler bara kan skrivas med VERSALER. Utöver detta måste skapandet av motif ske i motifblocket och ett motif kan inte tilldelas ett annat motif. Kontrollstrukturer såsom if-, repeat- och for-satser kan också bara skrivas i motifblocket.

I segmentblocket skrivs motif över till segmentvariabler och slutligen i strukturblocket returneras segmenten i den valda ordningen. Songic kod kan med den nuvarande implementationen bara returnera notsekvenser från filer med ändelsen .song.

2.7 MakeTest

MakeTest klassen finns i filen `make_test.rb`. Från den genereras och exekveras filen `Tests_Songic.rb` innehållandes testerna för Songic utifrån filnamnen och deras förväntade output i filen `expected_outputs.txt`. De genererade testerna skapas utifrån mallen `Template.rb` i katalogen `ztests`. Av testerna genereras ett ruby `TypeError` *Trying to access a non-existent variable* vilket är meningen.

2.8 Testgenerering

För att nya implementationer i Songic lätt ska kunna testas levereras språket med ett enkelt system att automatiskt generera och exekvera en testfil. Bara vetskapen om hur en terminal öppnas i projektkatalogen behövs. Filen i projektets sökväg `/tdp019/ztests/expected_outputs.txt` är uppbyggd av två kolumner. Namnet på det specifika testet och vilken fil som testas representeras av kolumnen till vänster. Det resultat som denna fil förväntas returnera återges av kolumnen till höger. Som exempel förväntas filen `basic_test.song` ha `a b` som retur och det specifika testet namnges `test_basic_test` i testfilen `Tests_Songic.rb`.

För att ett nytt test ska läggas till behövs en `ny_test_fil.song` först skapas i `/ztests` katalogen. Sedan skrivs namnet `ny_test_fil` in någonstans i den högra kolumnen av `expected_outputs.txt` och vad testet förväntas returnera på samma rad i den högra kolumnen. Testfilen körs genom att öppna en terminal i `/tdp019` katalogen och sen skriva `ruby make_test.rb`.

2.9 Instruktioner för installation

1. Klona repot från https://gitlab.liu.se/danhu849/tdp019/-/blob/master2/rules.rb
2. Skriv `ruby make_test.rb` i terminalen för att se att samtliga tester exekveras.
3. Filer ska ha filändelsen .song
4. Utgå ifrån mallen nedan när nya program skrivs.
5. Exekvera Songicprogrammet i terminalen med kommmandot: `ruby songic.rb nytt_fil_namn.song`

Songicprogram mall:

```
motif{}
segments{}
structure{}
```

2.10 Reflektion

Det största misstagen som vi kan först se nu i efterhand är att språkets design inte anpassats efter betygskriterierna. Därav har funktionalitet för språket implementerats även om dessa ej skulle användas av en potentiell användare. I slutet kändes det konstigt att skapa något vi visste om aldrig skulle bli användbart.

Det andra misstaget blev att vi kom så långt i utvecklingen av att använda parsern också som evaluerare av vår kod. Domänen språket styrde över var för liten och det tog lång tid innan vi fick vetskapen om att något var fel. Efter att grunden för Songic språket lagts och de mest basala funktionerna skrivits in behövdes hela grunden skrivas om. Detta behövdes olyckligtvis göras två gånger. Första gången lades en Rotnodsklass till och parsern skrevs om till att lägga alla objekt i parsern. Istället för att evaluera rotnoden utgjordes Noteklassens .seval funktion av att skriva ut sig själv i terminalen med print. Därför gjordes ingen förflyttning av data och felet upptäcktes inte förrän vi började med utvecklingen av scope. I och med utvecklingen av scope upptäcktes det att minneshanteringen inte sköttes ordentligt så när klassen Stack lades till behövde kodbasen skrivas om igen.

Ett stort problem var att utskriften av alla olika implementationer blev det förväntade resultatet och därför tog det lång tid att hitta felen samt att äldre onödig eller obrukbar kod oftast låg kvar mellan versionerna utan att vi märkte det.

Hade vi haft förståelsen för rotnodens betydelse tidigare och förstått dess funktion och roll i språket hade vi kunnat lägga till mycket mer funktionalitet och förmodligen tagit kortare tid att lägga till scope. Då koden behövde skrivas om ett par gånger kom vi också igång senare än förväntat med testerna. Att inte kunna utveckla med tester på grund av att vi inte visste vad eller hur resultatet skulle returneras försvårade utvecklingsprocessen.

Efter att kunskapen av hur returen såg ut gick det snabbt att sätta upp testsystemet och skriva klart resterande tester. Med den klarhet och insikt vi fått från projektet tänker vi oss att samma uppgift skulle kunna klaras av på en bråkdel av vad det tagit oss nu.

Kortfattat upplever vi ändå att språket och dess implementering har lagt en bra grund för vidare utbyggnad då det finns delar i implementationen som tagit i höjd för framtida utveckling även om de inte används i sin nuvarande form.