LINKÖPINGS UNIVERSITET

TDP019 Projekt: Datorspråk

Systemdokumentation

Författare

Warren Crutcher, warcr701@student.liu.se
Viktor Norlin, vikno856@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2023
Version 1.2

2023-05-25

Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.2	Uppdatering av grammatik och funktionalitet	2023-05-25
1.1	Reviderad version av första utkast	2023-05-24
1.0	Första utkast av dokumentation	2023-05-04

Innehåll

Tabeller

1 Inledning

I kursen TDP019 Projekt: Datorspråk som läses under andra terminen av Innovativ Programmering vid Linköpings universitet har vi skapat ett programmeringsspråk som heter Whisper.

1.1 Syfte och målgrupp

Syftet med Whisper är att underlätta hanteringen av enkla matematiska uttryck i terminalen. Målgruppen för Whisper är användare som är bekanta med en Unix-terminal men inte nödvändigtvis är erfarna programmerare som önskar en miniräknare med användardefinierade funktioner i terminalen.

2 Användarhandledning

Whisper fungerar som en terminalbaserad miniräknare. Först måste användaren ladda ner filerna från gitlab. Kopiera filerna med kommandot:

```
$ git clone https://gitlab.liu.se/warcr701/TDP019.git
```

För att installera Whisper, kör installationsskriptet med kommandot:

```
$ sudo ./whisper_install
```

Detta gör att Whisper installeras och kan köras som ett exekverbart program vart som helst i terminalen efter att ha startat om datorn. För att sedan börja använda Whisper, kör:

```
$ whisper
```

För att få tips för att använda Whisper, kör:

```
$ man whisper
```

För att avsluta programmet kör:

```
[Whisper]: exit
```

Om installation ej önskas, kan användaren från den klonade katalogen köra:

```
$ ruby whisper/whisper.rb
```

Terminalen bör se ut såhär:

```
[Whisper]:
```

Nu kan användaren skriva olika matematiska uttryck som exekveras direkt i terminalen.

2.1 Syntax-exempel

Användaren kan skriva enkla vanliga matematiska uttryck, till exempel:

```
[Whisper]: 5+6*9
```

Whisper stödjer användarskapade aritmetiska uttryck. Exemplet nedan visar definiering av en funktion, variabel-definition, utskrift av ett funktionsanrop till terminalen och iteration genom en while-loop:

```
[Whisper]: let f(x,y) = {x*y + x^y}; let a = 0; while a < 10 do {show(f(a,2)); let a = a+1}
1
4
8
14
24
```

```
42
76
142
272
530
=> 10
[Whisper]:
```

I exemplet ovan skrevs resultatet av funktionsanrop f(a, 2) ut medans värdet för `a` ändrades under varje iteration av loopen. Sista utskriften vid `=>` ger resultatet av det sista exekverade uttrycket som kördes, nämligen `let a = a + 1`.

Whisper stödjer även att skicka en funktion som argument till en annan funktion:

```
[Whisper]: let f(x,y) = {x*y + y^x}; let g(x) = {x^3}; let a = 0; while a < 10 do \
           {show(f(g(a), 2)); a += 1}
```

Det finns två olika sätt att öka värden i en while-loop i Whisper. Uttrycket `let a = a + 1` fungerar som en variabel-deklaration som gör att enligt räckviddhanteringen (som kallas för scopehanteringen) deklareras `a` i nuvarande scope. Detta gör att `a` kommer att ha samma värde efter att kodblocket avslutas som innan. I exemplet ovan, skulle `a` ha värdet `0` när kodblocken avslutas. Med uttrycket `a += 1` kommer användaren undan scopehanteringens regler så att `a` kommer ha värdet som den hade inuti kodblocken efter att kodblocken exekverats. På ett sådant sätt kan användaren välja hur de vill hantera variabler inuti loopar.

Funktioner kan också skapas inuti andra funktioner. Exempel:

```
[Whisper]: let f(x, y )  = { if x < 10 then {x^y} else {let g(c) = {c+2}; g(x)}}
[Whisper]: f(10, 3)
```

Anropas `f` med ett värde större än eller lika med `10` kommer else-satsen exekveras där funktionen `g` skapas och exekveras med `x`-värdet som argument. Resultatet efter evalueringen av exemplet ovan ger oss:

```
=> 12
```

vilket förväntas.

2.2 Sammanfattning av syntaxregler

De grundläggande satserna som stöds i Whisper är: variabler, funktioner (med rekursion), if-satser, while-loopar, arrayer, for-loopar, inbyggda array funktioner, en hash-linknande datatyp som heter *Record* och ett antal artimetiska och logiska operationer.
Nedan följer en sammanfattning av de ovannämnda satserna.

2.2.1 Variabler

Deklaration av variabler, d.v.s. tilldelningssatser:

```
let <variable name> = <value>
```

Där `<variable name>` måste börja med en liten bokstav och bara får innehålla bokstäver, siffror eller `_` tecken.

2.2.2 Funktioner

Funktioner definieras på följande sätt:

```
let <function name>() = {<statement>}
let <function name>() = {<statement_1>; ...; <statement_i>}
```

```
let <function name>(<argument>) = {<statement>}
let <function name>(<arg_1>, ..., <arg_i>) = {<statement_1>; ...; <statement_k>}
```

där både antalet argument och efterföljande satser är av godtycklig mängd. Namnet på en funktion följer samma regler som namnet för en variabel men en funktion utan argument måste följas av `()` och högersidan av operatorn `=` måste börja med `{` och sluta med `}`, d.v.s. en eller flera `<statement>` måste vara inuti `{}` och flera `<statement>` måste separeras med `;`-tecken.

2.2.3 Rekursion

Whisper stödjer rekursiva funktioner. Användaren kan till exempel skriva och exekvera:

```
[Whisper]: let factorial(n) = {if n <= 0 then {1}
else {n * factorial(n-1)}}; factorial(7)
=> 5040
```

Att kunde använda rekursion bidrar till Whispers syfte av att förenkla aritmetiska beräkningar. Ett till exempel:

```
[Whisper]: let fib(x) = {if x <= 0 then {0} else
{if x == 1 then {1} else {fib(x-2) + fib(x-1)}}}; fib(11)
=> 89
```

Exemplet ovan visar också hur nästlade villkorsatser ser ut i Whisper.

2.2.4 If-satser

Whisper har stöd för if-satser och if-else-satser.

```
if {<condition>} then {<statements>}
if {<condition>} then {<statements>} else {<statements>}
```

Uttrycket för `<condition>` måste returnera antingen sant eller falskt vid evaluering och `<statements>` kan vara en eller flera satser separerade med semikolon.

2.2.5 While-loopar

Whisper har stöd för while-loopar för att uppnå iteration och olika sätt att styra ett programs struktur.

```
while {<condition>} do {<statements>}
```

Precis som beskrevs med if-satser (se avsnitt 2.2.3) måste `<condition>` vara en sats som evalueras till antingen sant eller falskt och `<statements>` kan vara en eller flera semikolon-separerade satser.

2.2.6 Användarskapade arrayer

En array i Whisper är en kommaseparerad lista av en godtycklig mängd värden inom hakparenteser.

```
[1,2,3,4,5..., n]
```

Arrayer i Whisper deklareras på samma sätt som funktioner och andra variabler.

```
let <variable name> = <array>
```

Arrayer kan användas i både addition- och subtraktionoperationer.

```
[1,2,3] + [4,5]
```

returnerar `[1,2,3,4,5]` och `[1,2,3,4,5] - [1,5]` returnerar `[2,3,4]`.

2.2.7 Inbyggda arrayer

Det finns två inbyggda arrayer i Whisper, `N{n}` och `Primes{n}`. Array `N{n}` innehåller alltid `0` som första element och sedan alla konsekutiva positiva heltal till och med `n`. Array `Primes{n}` innehåller alla primtal upp till värdet `n`.

```
N{10} = [0,1,2,3,4,5,6,7,8,9,10]
Primes{20} = [2,3,5,7,11,13,17,19]
```

Inbyggda arrayer användas på samma sätt som andra arrayer.

```
let n = N{3}; let p = Primes{15}; n + p
=> [0,1,2,3,5,7,11,13]
```

2.2.8 For-loopar

En for-loop används i Whisper för att iterera över arrayer.

```
for x in <array> do {<statements>}
```

I exemplet ovan kan `<array>` vara antingen en variabel som representerar en array eller en array *literal* som `[1,2,3,4,5]`. Exemplet nedan visar hur alla tal i en array kan enkelt skrivas ut.

```
for x in [1,3,5,7,9] do {show(x)}
1
3
5
7
9
=>
```

2.2.9 Inbyggda array-funktioner

Whisper har sju inbyggda array-funktioner, `empty`, `size`, `sort`, `flip`, `set`, `map` och `filter`. Exemplet nedan visar dess användning.

```
let a = [2,3,1,1,4]; let triple(x) = {3*x}; let mod_2(x) = {x%2 == 0}
=>
a?empty
=> false
a?size
=> 5
a!sort
=> [1, 1, 2, 3, 4]
a!flip
=> [4, 3, 2, 1, 1]
a!set
=> [1, 2, 3, 4]
let a = a!set
=> [1, 2, 3, 4]
let a = a.map(triple)
=> [3, 6, 9, 12]
a.filter(mod_2)
=> [6, 12]
```

Nedan finns en sammanfattning av de sju inbyggda array-funktionerna.

- `empty` returnerar `true` om arrayen innehåller null antal element, annars returneras `false` .
- `size` returnerar antal element i arrayen.
- `sort` sorterar arrayen i stigande storleksordning.
- `flip` vänder ordningen på elementen i arrayen.
- `set` sorterar arrayen och minskar antalet element i arrayen för att endast innehålla unika element.
- `map` tar en funktion som argument och exekverar funktionen på varje element i arrayen.
- `filter` tar en funktion som argument och tar bort alla element där funktionen returnerar `false`.

2.2.10 Record

Whisper har en hash-liknande datatyp som heter Record. En Record är en associativ databehållare, där de olika värdena som finns i en Record anropas med sitt namn istället för en index av en array.

```
let Person_1 = {height = 180; weight = 75};
Person_1.height
=> 180
Person_1.weight
=> 75
```

Record har inte mycket funktionalitet i nuläget och finns bara för att tillåta användaren att samla data på ett associativt sätt. En diskussion om begränsningar med användning av Record finns i avsnitt 4.1.

2.2.11 Operationer

Whisper har stöd till ett begränsat antal grundläggande aritmetiska operationer. Se tabell 1.

Tabell 1: Binära och unära operationer som stöds av Whisper

Operation	**Symbol**
Binära operatorer	
bitvis vänster flytt	`<<`
bitvis höger flytt	`>>`
bitvis och	`&`
bitvis XOR	`^`
bitvis eller	`|`
addition	`+`
subtraktion	`-`
multiplikation	`*`
division	`/`
modulär division	%
exponentiering	`@`
omfördela addition	`+=`
omfördela subtraktion	`-=`
omfördela multiplikation	`*=`
omfördela division	`/=`
och	`&&`
eller	`||`
Jämförelse-operatorer	
mindre än	`<`
större än	`>`
mindre än eller lika med	`<=`
större än eller lika med	`>=`
lika med	`==`
ej lika med	`!=`
Unära operatorer	
ej	`!`
negation	`-`
bitvis negation	`~`

3 Systemdokumentation

Nedan beskrivs Whisper-språket i sin helhet.

3.1 Översikt

Språket Whisper ligger ovanpå den lexikaliska scannern som används i kursen TDP007: Konstruktion av Datorspråk. Den lexikaliska parsern är skapad så att användaren kan skriva olika *regler* som exekveras beroende på nyckelord eller reguljära uttryck som läses av scannern. De olika regler som exekveras bygger ett *abstrakt syntaxträd* med olika uttryck kopplade till varandra som sedan exekveras.

De olika delarna som uppgör ett abstrakt syntaxträd består av olika klasser i språket Ruby. Whisper har också en klass som hanterar räckvidden av olika funktioner och variabler, oftast kallat *scopehantering*.

I och med att Whisper har som syfte att fungera som en miniräknare är det tänkt att det lämpligaste sättet att använda Whisper är direkt i en Unix-terminal i interaktivt läge där varje uttryck evalueras direkt för att ge användaren snabba svar. Nedan finns ytterligare beskrivning i mer detalj av de olika delarna som tillsammans ligger bakom Whispers konstruktion.

3.2 Grammatik

Whispers grammatik definieras enligt BNF (Backus-Naur Form) och finns i slutet av detta dokument i tabell 2.

3.3 Lexikalisk analys och parsing

Whisper är byggt på parsern som användes i kursen TDP007, som fanns i filen `rdparse.rb`. I Whisper-projektet finns koden i filen `parser.rb` och innehåller klasserna `Rule` och `Parser`. De här två klasserna innehåller tillsammans ramverket för en så kallad *recursive descent parser* och tillåter programmerare att skapa regler enligt en grammatik för att exekvera kod enligt programmerarens önskemål. I Whisper-projektet har författarna inte ändrat varken `Rule`- eller `Parser`-klasserna.

3.4 Evaluering

Filen `whisper.rb` innehåller en implementation av vår BNF-grammatik som skapar instanser av olika klasser beroende på vilka regler som matchas med parsingen av filen `parser.rb`. När en matchning av en sats inträffar, skapas ett abstrakt syntaxträd vars kod finns i filen `nodes.rb` där varje nod-klass har en egen uppbyggnad och `eval`-funktion. På grund av Whispers betoning på interaktivitet skapas, evalueras och förstörs en trädstruktur för varje körning av ett Whisper-program. Ett Whisper-program består av antingen en sats (statement) eller flera satser (en lista av statements). I och med att varje nod-klass är ett abstrakt syntaxträd för sig så länkas olika abstrakta syntaxträd ihop, där vissa satser blir 'barn'-satser till olika abstrakta syntaxträd. En 'körning' av ett Whisper-program räknas som när användaren trycker på 'Enter'-tangenten på tangentbordet.

För att Whisper ska komma ihåg variabel- och funktionsdefinitioner mellan olika körningar under en session finns en klass som heter `scope.rb` som innehåller klassen `Scope_Manager`. `Scope_Manager`s uppgift är att förvara definitioner av olika funktioner och variabler samt vilket värde de olika 'symbolerna' för både funktioner och variabler representerar.

3.5 Klasser

I och med att Whisper implementeras i det objektorienterade programmeringsspråket Ruby används klasser som egendefinierade datatyper. Det som klasserna hanterar är omfattning (scoping), parsing, konstruktion och evalueringen av uttryck i ett abstrakt syntaxträd (AST).

3.5.1 Scopehanteringen

Klassen `Scope_Manager` hanterar en variabels eller funktions omfattning i Whisper. Sättet som `Scope_Manager` fungerar är som en hög (stack) av tabeller där varje tabell har ett nyckelvärde som namn till en funktion eller variabel och har som värde en referens till ett AST-objekt som representerar det givna uttrycket. I Ruby uppnås det här beteende med en datamedlem till `Scope_Manager` som heter `scope` som är en `Array` av `Hash`-tabeller. `Scope_Manager` är statisk, d.v.s. att det finns bara en instans av `Scope_Manager` i Whisper och den nås av alla andra klasser. När programmet går in till en ny nivå av scope läggs en ny `Hash` till i `Scope`-arrayen i `Scope_Manager`. Klassen har funktionen `symbol_lookup()` som hämtar värdet av en symbol. Funktionen börjar alltid leta i nuvarande nivå på stacken för att hitta värdet av en symbol. Om symbolen inte finns fortsätter sökningen i de andra nivåer tills variabelns värde hittas. Annars returnerar funktionen `nil`.

Förutom funktioner för att lägga till eller ta bort en omfattningsnivå finns en funktion `symbol_update()` som ändrar värdet av en symbol oavsett nuvarande omfattningsnivå. På ett sådant sätt kan en variabel ändras inuti en loop eller annat kodblock och beroende på användarens önskemål antingen behåller dåvarande värde eller uppdateras även efter kodblocket exekverats.

3.5.2 AST-Noder

Förutom klasserna som tillhör parsern som finns i filen `parser.rb` finns flera olika nod-klasser som finns inuti filen `nodes.rb`. Dessa är:

- `Node_Arithmetic` som representerar ett abstrakt syntaxträd med en operation och ett eller två *barn*-noder beroende på om operatorn är binär eller unär.
- `Node_Array` som representerar ett absrakt syntaxträd med attribut `@expresions` som är en lista av AST-noder och `@size` som är antal element i `@xpressions`.
- `Node_Array_Access` som är ett AST med ett `@index`-värde för att leta upp olika element i ett `Node_Array.expressions`-objekt.
- `Node_Array_Assignment` som är ett AST med funktionalitet för att kunna ändra individuella element i ett `Node_Array.expressions`-objekt. `Node_Array_Assignment` tillåter operationer som `a[0] = 9` för en array `a`.
- `Node_Array_Naturals` ärver från `Node_Array`. En `Node_Array_Naturals`-nod har en annan konstruktor än ett `Node_Array`-objekt. Detta eftersom att det är bestämt i förväg vilka element som ska ingå i `@expressions` attribut av ett `Node_Array_Naturals`-objekt, nämligen `0` och heltal till och med ett användarangivet tal `n`.
- `Node_Array_Primes` ärver också från `Node_Array` men innehåller alla primtal upp till ett användargivit tal `n`.
- `Node_Assignment` som representerar ett abstrakt syntaxträd med operatorn `=` som också för in nya symboler till `Scope_Manager`-klassens `stack` av symbol-tabeller.
- `Node_Boolean` som representerar ett boolskt uttryck av antingen nyckeloret 'true' eller 'false'.
- `Node_Break` är en AST-nod med inga barn som representerar uttrycket 'break' för att avsluta iterationer inuti en loop.
- `Node_Built_In_Function` som representerar ett abstrakt syntaxträd för inbyggda funktioner. I nuläget har Whisper stöd för bara nio inbyggda funktioner. Nedan finns en lista med varje inbyggda funktion med `<argument_type> -> <return_type>` bredvid för argumenttyp och returtyp som funktionen använder.
  - `show   :: any     -> nil`
  - `prime  :: integer -> boolean`
  - `empty  :: array   -> boolean`
  - `size   :: array   -> integer`
  - `sort   :: array   -> array`
  - `flip   :: array   -> array`
  - `set    :: array   -> array`
  - `map    :: array, function -> array`
  - `filter :: array, function -> array`

  De nio inbyggda funktionerna finns beskrivit i mer detalj i kodens dokumentation. `Node_Built_In_Function` använder Rubys `case - when` syntax för att bestämma hur noden evalueras beroende på vilken funktion användaren anropar.
- `Node_Comparison` som representerar ett AST med jämförelseoperatorer, som `< > <= >= ==` och `!=`.

- `Node_Continue` representerar en AST-nod utan barn-noder för att representera uttrycket *continue*, som kan användas av programmerare för att förtydliga sina avsikter att programmet ska fortsätta utan att göra något.
- `Node_Double` som representerar ett AST för ett värde som kan vara ett decimaltal. Noden har inga barn.
- `Node_For_Each` som representerar en AST-nod för iteration över en array. Noden har som attribut `@iterator` som är en variabel som representerar varje värde i en array under iteration. `@expression` är den arrayen som ska itereras över och `@body` är ett kodblock som ska exekveras under varje iteration.
- `Node_Function_Call` som representerar ett AST för hanteringen av ett funktionsanrop. Klassen använder också `Scope_Manager`-klassen för att lägga till en ny omfattningsnivå vid funktionskroppsstart och omfattningsnivån tas bort efter evalueringen avslutas.
- `Node_Function_Define` som representerar ett AST för hanteringen av en funktionsdefinition. Evalueringen består av att lägga funktionsdefinition i symbol-tabellen i nuvarande omfattningsnivå.
- `Node_If` som representerar ett AST för `if`-satser. Den här noden har som barn ett AST som är en `condition`, en annan AST-nods `then_block` för uttryck som evalueras ifall `condition` är sann och antingen ett barn som är `nil` eller om en `else`-sats finns en AST-nod som heter `else_block` som evalueras om `condition` evalueras till falskt.
- `Node_Integer` som representerar ett heltal som AST.
- `Node_List` som representerar ett AST och ärver från Rubys `Array`-klass. En `Node_List` har som syfte att hantera antingen en lista av argument eller en lista av uttryck som ska evalueras. Att `Node_List` ärver från `Array` är för att ta nytta av de olika inbyggda funktioner som ingår i Rubys `Array`-klass.
- `Node_Logical_Op` som representerar ett AST för logiska operationer, som `&&` || `!` och `==`.
- `Node_Reassign` som representerar ett AST för operationer `+= -= *= /=`. De här operationerna undviker `Scope_Manager`-klassens hantering av tilldelningsoperationer.
- `Node_Record` representerar ett AST för den associativa behållaren Record. Ett `Node_Record`-objekt har attributet `@fields` som är en samling av variabelnamn och deras värde.
- `Node_Record_Access` har som attribut `@record_name` och `field_name` för att använda hämta ett värde av ett *fält* som lagras i en `Node_Record`s `@fields`.
- `Node_Show` hanterar anrop till `show`-funktionen som egen nod. Detta görs eftersom att beteendet av evalueringen innan ett värde skrivs ut kan variera beroende på den underliggande Ruby-datatypen.
- `Node_Variable` som representerar ett AST för ett variabelnamn. Evalueringen av en `Node_Variable`-instans består av hämtning av den AST-noden som innehåller uttrycket med samma namn som variabelns instans.
- `Node_While` som representerar ett AST för en `while`-loop. En `Node_While` har som barn-noder ett AST för `condition` och `block` där noden i `block` evalueras så länge satsen i `condition` returnerar sant.

3.6 Kodstandard

Whisper är skapad för att ha en minimal syntax. I grammatiken och kodexemplaren så framgår det att det inte finns många nyckelord. Tanken är att ett program är ett uttryck eller en sekvens av uttryck och förutom semikolon som används för att skilja mellan olika uttryck bryr Whisper inte om ett uttryck ser ut som

```
let          a =                                          18
```

eller

```
let a=18
```

så länge att nyckelorden matchas. Användning av nyckelord `then` och `do` är mer för att ge ett 'mänskligt' sätt att beskriva instruktioner.

3.7 Paketeringen av koden

Som beskrivs i avsnitt 3 så kan Whisper installeras så att den finns exekverbar vart som helst i en Unix-terminal, eller så laddas de givna filerna ned och i katalogen som innehåller filen `whisper.rb` körs kommandot:

```
$ ruby whisper/whisper.rb
```

Terminalen borde visa:

```
[Whisper]:
```

Då kan användaren börja skriva uttryck som beskrivs i avsnitt 2. För att installera Whisper till systemet, kör:

```
$ sudo ./whisper_install
```

Efter installationen borde Whisper vara tillgänglig i terminalen med en medföljande manualsida. Datorn måste dock startas om efter installationen på grund av uppdateringen av `$PATH` variabeln i `.bashrc` filen.

4 Begränsningar

Trots funktionaliteten som finns i Whisper finns begränsningar med språkets implementation.

4.1 Records

Datatypen Record var ett sent försök att implementera en associativ databehållare. Författarna är medvetna om begränsningarna med implementationen av Record och vill därför upplysa läsaren.

4.1.1 Parsing av Record-fält

Problemet just nu är att bestämma bästa sättet att hantera Record-fält som är asymmetriska. Under parsing sparas fält i arrayer av par, d.v.s `[[name1, value1], [name2, value2], ...` och med ett jämnt antal par fungerar parsing som förväntat. Med udda antal par blir det problem. Programmet kommer krascha tills vi har implementerat ett annat sätt. Vi inkluderar Record datatypen för att den ingår i projektets nuvarande version men upplyser för användaren att den är inte särskilt användbar i nuläget.

4.1.2 Felhantering

Felhanteringen i Whisper är svag. Syntaxfel som uppstår under lexikalisk analys ger väldigt ohjälpsamma meddelande. Författarna har inte ändrat något i parsern `parser.rb` som användes i kursen TDP007. Några felmeddelande under programmets *runtime* ges vid fel användning av datatyper eller användning av odeklarerade variabelnamn. Däremot är felhanteringen långt ifrån robust och vid ytterligare utveckling av språket skulle implementeringen av robusta felhantering ha hög prioritet.

Författarna vill vara framkomliga med att de är väldigt medvetna om begränsningarna med språket och är öppna på alla förslag för att förbättra implementeringen.

5 Erfarenheter och reflektion

Innan projektet drog i gång så var gruppens båda medlemmar medvetna om att ambitionerna bör hållas på en låg nivå, då projektets omfattning är relativt stort. Vi hade aningar om att vi skulle stöta på problem som skulle göra att utvecklingen tog längre tid än tänkt. Trots att våra ambitioner var relativt låga redan innan projektet började så var vi tvungna att ytterligare sänka våra krav en kort tid efter projektet dragit i gång. Från början var tanken att vårt språk skulle vara statiskt typat och kompilerat, något som vi gav upp på omgående då det kändes som en alldeles för svår uppgift.

Det rådde stor frustration i gruppen under stora delar av projektets gång. De första veckorna av projektet lades mycket arbete på andra kurser, vilket gjorde att projektet fick lägre prioritet. När väl projektet var i fokus så hade gruppen svårt att komma i gång och få ner någon konkret och användbar kod. Både Viktor och Warren kände att avsaknaden av dokumentation kopplad till rdparse försvårade arbetet markant. Detta gjorde att Warren på egen hand under sin fritid utvecklade en alternativ version av programspråket med hjälp av verktygen Flex och Bison i C. När gruppen hade två versioner i två olika verktyg så var det svårt välja en version av projektet att gå vidare med. Dessutom så gick utvecklingen i Flex/Bison vid denna tidpunkt snabbare än utvecklingen i rdparse. Detta var till stor del en följd av att gruppen inte än kommit fram till en bra struktur över hur parsingen ska gå till i rdparse. Under denna period så var båda gruppens medlemmar frustrerade och förvirrade, och moralen i gruppen var inte särskilt bra. Efter ett antal veckor fyllda med frustration så kom gruppen fram till en metod som löser parsingen i rdparse och ger ett förväntat resultat. När denna metod var på plats så gick utvecklingen framåt i rask takt i jämförelse med innan. Med facit i hand så hade gruppens initiala plan troligtvis gått att genomföra om inte flera veckors arbete hade gått till spillo på grund av osäkerhet gällande hur implementationen skulle genomföras.

Det har varit en otrolig lärorik upplevelse att gräva djupt i ämnena parsning, sammanhangsfria grammatiker och trädutvärdering. Åtminstone en av gruppmedlemmarna är, efter att ha känt sig något besegrad av rdparse.rb, väldigt intresserad av att lära sig hur man bygger sin egen parser för ett enkelt språk som Whisper. Även om det är uppenbart för oss att vi inte hade varit redo att göra vår egen lexer och parser när kursen började känns det som ett uppnåeligt mål nu, med tanke på den enorma tillväxt som har skett.

Gruppdynamiken var positiv. Vi båda tyckte om att arbeta tillsammans och njöt av varandras sällskap. Men ibland kändes det som att detta hindrade oss från att fatta beslut om vi var oense i vilken riktning projektet skulle ta. Ingen av gruppmedlemmarna ville bestämma över den andra. Detta är något att vara medveten om för framtida projekt. Sammantaget har det varit ett utmanande projekt och kommer med största sannolikhet att fortsätta som ett hobbyprojekt av en av gruppmedlemmarna.

Tabell 2: Whisper Grammatik.

Whisper Grammatik		
<program>	::=	<statement_list>
<statement_list>	::=	<statement> | <statement_list> ';' <statement>
<statement>	::=	<built_in_evaluate> | <if_statement> | <while_statement> | <function_define> | <assignment_statement> | <for_each_statement> | <break_statement> | <continue_statement> | <expression>
<built_in_evaluate>	::=	'show' '(' <expression> ')' <expression> '?' <inquiry> | <expression> '?' <command> | <expression> ' . ' <array_transform> '(' <id> ')'
<inquiry>	::=	'empty' | 'size'
<command>	::=	'sort' | 'flip' | 'set'
<array_transform>	::=	'map' | 'filter'
<if_statement>	::=	'if' <expression> 'then' '{' <statement_list> '}' 'else' '{' <statement_list> '}' | 'if' <expression> 'then' '{' <statement_list> '}'
<while_statement>	::=	'while' <expression> 'do' '{' <statement_list> '}'
<function_define>	::=	'let' <id> '(' ')' '=' '{' <statement_list> '}' | 'let' <id> '(' <argument_list> ')' '=' '{' <statement_list> '}'
<argument_list>	::=	<id> | <argument_list> ',' <id>
<assignment_statement>	::=	'let' <variable> '=' <built_in_evaluate> | 'let' <id> '=' <expression> | <variable> <re_assign_op> <expression> | 'let' <variable> '[' <expression> ']' '=' <expression> | 'let' <record_name> '=' <record_fields>
<for_each_statement>	::=	'for' <id> 'in' <expression> 'do' '{' <statement_list> '}'
<break_statement>	::=	'break'
<continue_statement>	::=	'continue'
<record_fields>	::=	<id> '=' <expression> | <record_fields> ';' <id> '=' <expression>
<id>	::=	`/[a-z\_][a-zA-Z0-9\_]*/`
<function_evaluate>	::=	<id> '(' ')' | <id> '(' <explist> ')'

<explist>	::=	<expression> | <explist> ',' <expression>
<expression>	::=	<record_access> | <term> | <variable> | <continue_statement> | <break_statement>
<term>	::=	<logical_term> | <arithmetic_term>
<arithmetic_term>	::=	<arithmetic_factor> | <arithmetic_term> <addop> <arithmetic_factor> | <arithmetic_term> <rel_op> <arithmetic_factor>
<arithmetic_factor>	::=	<arithmetic_exponential> | <arithmetic_factor> <bitwise_op> <arithmetic_exponential> | <arithmetic_factor> <mulop> <arithmetic_exponential> | <arithmetic_factor> <eq_op> <arithmetic_atom>
<arithmetic_exponential>	::=	<array_access> | <arithmetic_exponential> <ex_op> <array_access>
<arithmetic_atom>	::=	'Primes' '{' <digit> '}' | <function_evaluate> | <literal> | '(' <arithmetic_term> ')' | <unary_minus> <arithmetic_atom> | '{' <record_fields> '}'
<logical_term>	::=	<logical_factor> | <logical_term> <or_op> <logical_factor>
<logical_factor>	::=	<logical_atom> | <logical_factor> <andop> <logical_atom>
<logical_atom>	::=	<bool> | <literal> '?' 'prime' | '(' <logical_term> ')' | '!' <logical_atom>
<re_assign_op>	::=	<increment> | <decrement> | <re_divide> | <re_multiply>
<unary_minus>	::=	/\-/ | /\~/
<increment>	::=	/\+\=/
<decrement>	::=	/\-\=/
<re_divide>	::=	/\/\=/
<re_multiply>	::=	/*\=/
<bitwise_op>	::=	/\<\</ | /\>\>/ | /\&/ | /\^/ | /|/

<add_op>	::=	/\+/ | /\-/
<mul_op>	::=	/*/ | /\// | /\%/
<ex_op>	::=	/\^/
<or_op>	::=	/'||'/
<and_op>	::=	/'&&'/
<rel_op>	::=	/\>/ | /\</ | /\<\=/ | /\>\=/
<eq_op>	::=	/\=\=/ | /\!\=/
<variable>	::=	/[a-z_][a-z_0-9]*/
<literal>	::=	<digit> | <variable>
<digit>	::=	/\-?\.?[0-9]+/ | /\-?[0-9]+\.[0-9]*/
<bool>	::=	'true' | 'false'
<record_access>	::=	<record_name> ' . ' <id> | <record_access> ' . ' <id>
<record_name>	::=	/[A-Z][a-zA-Z0-9_]+/