Framsida

EasyToRead är vårat språk, och det är skapat för precis vad det låter som: lättläslighet.
Vi heter Max och Markus, och vi ska berätta om hur vi har utvecklat detta språk.

Tanken Bakom

Den ursprungliga tanken var att skapa ett språk som var så lättläsligt som möjligt, så att personer runt vår ålder som inte har programmerat (mycket) tidigare skulle få det enklare att vänja sig vid tankarna bakom kodning.
Därför så har vi utvecklat språket för folk i just den positionen. Vi tänkte tillbaka till vår första termin och att vi lärde oss imperativ programmering. Detta använde vi för att utveckla tanken att språket ska användas imperativt av nybörjare (helst med hjälp av instruktör), där de lär sig variabelhantering och funktionsskapande.
Tanken var även att lärarna skulle utveckla klasser som eleverna skulle använda under lektioner och labbar, vilka är specifika för just den lektionen/labben.

Språket är även interpreterat, vilket är enklare att förstå och använda för en nybörjare inom kodning.
Vi tyckte att detta var en bra idé då vi aldrig hade hört talas om ett språk som är utvecklat runt just detta område. COBOL är ju till exempel designat för att vara lätt att läsa, men det blir inte lätt att förstå.

Kodexempel

”Hello World”-funktion

Här ser vi en utveckling av det första man alltid gör i ett programmeringsspråk: ”Hello World”-programmet.
Detta är, som ni ser, utvecklat till att vara en funktion som skriver ut ”Hello World” på en ny rad.

Det man kan lära sig från detta kodexempel är dels hur man definierar en funktion, att vi har implementerat semikolon som separator, använder oss av måsvingar för kodblock, och hur man kallar på en funktion efter definition.
Som ni ser så kan man kalla på en funktion genom att byta ut alla understreck i ett funktionsnamn med mellanslag.

Funktion med parametrar

Här går vi in lite djupare och visar hur man använder sig av parametrar i ETR.
I funktionsnamnet så lägger vi till symboler som innehåller ett dollartecken och ett tal. Denna symbol representerar var du sätter in den parametern vid funktionsanropet.
Alltså: som ni ser vid funktionsanropet så anropar vi funktionen som ”func false 'str';”, där ”false” och ”'str'” representerar parameter 1 och 2, vilket knyter samband med symbolernas nummer.

Man kan även se hur man hämtar ut värdet ur en variabel, genom användning av apostrofer. Detta, för att gå in på den mer tekniska sidan, hämtar ut värdet ur ”self”-variabeln i instansobjektet (vilket i sin tur hämtar ut värdet i sin ”self”, osv.).
Om man är observant kan man även se att vi sätter kolon runt str-variabeln. Detta är till för att visa att det är från denna variabel vi anropar en funktion.

Klasser och klassfunktioner
Här definierar vi alltså en klass vid namn ”myclass”, vilket har en variabel ”self” och en funktion ”initialize_$n”. ”$n”-symbolen är unik för klassfunktioner, då det visar var namnet på objektet som vi anropar funktionen på ska ligga. Om du vill anropa en medlemsfunktion inifrån en annan medlemsfunktion av samma klass så låter du kolonen vara tomma (dvs. ”::”).
I ETR så är alla medlemsvariabler privata (då det inte passade in i vår syntax) och alla klassfunktioner publika.

”initialize_$n”-funktionen initierar alltså instansen av myclass som vi skapar nedanför. Det gör den genom att sätta ”self”-variabeln till 0.

Sedan så skriver vi ut värdet i ”self”-variabeln hos ”myobj” genom att, som sagt tidigare, använda oss av apostrofer.
Vi anropar även den inbyggda funktionen ”puts”, vilket helt enkelt skriver ut ett nyradstecken.

Designval

Eftersom tanken bakom ETR var att vara så lättläst, och den ”slappa” delen av syntaxen ligger i funktionsanrop, så betyder det att för att få ett så ”slappt” språk som möjligt måste man implementera många funktioner som inkluderar alla sätt du skulle vilja skriva ditt program. Att funktionsanrop är så generella är ju en fördel, men även en nackdel i detta omfång.

All vår hårdkodad syntax, så som att sätta variabelvärden, skriva ut saker på skärmen, etc. är väldigt grundläggande i det att de liknar andra språk väldigt mycket. Till exempel så är det grundläggande ”set-statementet” bara ”<variabelnamn> = <värde>”.

Vi bestämde oss även för att implementera måsvingar och semikolon i vårat språk, för att eftersom vi har så ”slappt” syntax så kan det kanske vara svårt att gå vidare till ett nytt, mer avancerat språk om ETR var det första du kodade i. Måsvingar och semikolon är där för att lätta övergången till ett annat språk.

Den första tanken med vårat språk var även att inte ha med apostrofer och kolon vid funktionsanrop, men vi fick det påpekat att man skulle behöva väldigt mycket mer proccessorkraft för att hitta den rätta funktionen som användaren ville använda. Därav implementerade vi dessa symboler för att definiera vad som var parametrar eller namn, och vad som var en del av funktionsnamnet.

Vi har även tänkt att ha så konsekvent syntax som möjligt, för att förenkla allt för användaren. Det är till exempel därför det ser likadant ut att definiera en variabel, en funktion och en klass. Detta är designat så för att alla definitioner kan bli ganska långa, men du behöver inte lära dig mer än ett syntax.

Implementation

Vi har valt att implementera ETR genom att använda två parsrar. Den första parsern (”readern”) läser in filen och sparar alla variabel-, funktions- och klassdefinitioner, men även if- och while-satser. Den andra parsern (”parsern”) används för att exekvera koden som den readern genererar. Parsern kräver ett mycket mer specifikt syntax som readern konverterar den inskrivna koden till.

Allt som readern sparar undan sparas i hållare, som till exempel klasshållare eller variabelhållare. Vi går snabbt igenom if- och whilehållarna strax.

Scopes är implementerade genom globala Hash och Array-variabler som innehåller alla hållare som readern sparat undan, men endast det som ska vara åtkomligt i detta scope.

De enda verktygen vi använde oss av var RDParse, och Ruby i allmänhet. Vi förlängde till exempel strängar ganska mycket, då det är det RDParse använder sig av mest.

När vi sparar undan if- och while-satser så sparas de undan som ”if 1”, osv., där ID:t representerar var i if-/while-arrayen just den if-satsen ligger. Detta fick vi inspiration för från Javas kompilerade kod, där de är sparade i liknande sätt. Vi implementerade det även för att det var enklare än att ge alla if-satser ett unikt namn, så att vi kunde lägga dem i en Hash.
ID:t skulle så klart kunna bli det unika namnet, men det var snabbare att hämta ut ett index ur en array än att hämta ut ett namn ur en Hash.

Kodutklipp

Det här är då exempel på hållare, vilka är if- och whilehållarna. Som ni ser innehåller ifhållaren tre variabler: content, logicstmt och othercont, medans whilehållaren bara innehåller två: content och logicstmt.
Content-variabeln i båda representerar vad som finns i huvudkodblocket. Logicstmt är det logiska uttrycket som determinerar om koden i kodblocket exekveras. Ifhållarens othercont är då vad som finns i ”else”-satsen, eller ”otherwise”-satsen som det heter i ETR.

De har även båda en ”equals?”-funktion som determinerar om de är likadana som en annan if- eller while-sats. Detta är för att de då kan använda det ID:t i respektive lista, vilket sparar in på RAM och proccessering.

Problem

Ett utav de största problemen vi hade var att vi inte visste hur vi skulle implementera syntaxträdet från parsern med hjälp av nodes (vilket nog var tanken, inte att använda en parser till). Detta har lett till många underproblem, till exempel att exekvering är mycket långsammare än i andra språk.

Vi hade även en tanke i början, att använda punkter som separator istället för semikolon som vi till slut valde. Detta val gjorde vi för att det skulle bli problem om man till exempel skulle skriva ”set x to 5.5 is greater than 10”, där 5.5 då skulle läsas som ett flyttal, och inte som två separata funktionsanrop.

Scopes var det näst största problemet, då Ruby inte duplicerar variabler automatiskt. Detta ledde till att variabler fick värden de inte skulle ha fått, för att du skickade in dem som parametrar och liknande.

Regex och RDParse i sig var även stora problem under en kort period, då vi fick stora problem att ett regex matchade till en halv token, vilket gjorde att RDParse gick in i det kodblocket, vilket i sin tur kastade en error för att den, till exempel, inte kunde hitta det variabelnamn den skickade in. Detta löste vi till slut genom att sluta kasta errors på de ställen där det hände, och returnerade ”nil” från kodblocket istället, vilket var kod för RDParse att gå vidare.

Felhantering var även det svårt i RDParse, då man till exempel inte kan kasta sina egna errors om RDParse inte kan matcha en token rätt, eller liknande. Detta löste vi genom att fånga alla errors, och även genom vår implementation av ett ”debug”-läge där allt som parsrarna gör skrivs ut till en fil vid namn ”debug.txt”, tillsammans med errorn och all kod. Detta är då tanken att man eventuellt ska skicka in denna fil till oss om man hittade en konstig bug i språket.

Arrayer har allt gått fel med. Saker har inte fått rätt värde, de blev inte utskrivna rätt, och massor av andra saker. Detta har lett till att arrayer helt enkelt inte är så användbara i vårat språk. Det som går att göra är att sätta in och hämta ut värden ur variabeln, och att jämföra den med andra arrayer. Men inte mycket mer...