Språkdokumentation

JULIET

Författare

Liam vanDoorn, liava796@student.liu.se
Hillevi Ragnemalm, hilra371@student.liu.se

2022-02-22

Innehåll

1 Inledning

I kursen TDP019 Projekt: Datorspråk för utbildningen Innovativ Programmering utvecklades Juliet, ett programmeringsspråk för dig som är intresserad av att utmana dig själv som programmerare men även som författare. Juliet låter dig väva in och skapa programmeringsuttryck i engelsk litteratur med hjälp av reserverade nyckelord och uttryck.

På så sätt kan användaren exekvera .txt-filer och använda allt som inte matchar Juliet-uttryck till sina egna kreativa verktyg och vi hoppas därför att få se en spännande korsning emellan litteratur och programmering.

2 Användarhandledning — ditt första Juliet-dokument

Juliet är ett litet programmeringspråk med grundläggande programmeringsmetoder. Vi hoppas därför att denna användarhandledning ger dig en övergripande förståelse för hur du kan skapa och exekvera ditt första Juliet-program.

För att använda Juliet till sin fulla potential krävs viss programmeringserfarenhet då det har konstruktioner som for-loopar, while-loopar och if-satser. Om användaren däremot endast använder Juliet för enkla uträkningar eller utskrifter krävs endast att korrekt syntax används.

2.1 Installation

Juliet kräver en installation av Ruby samt Juliets källkod, källkoden kan hittas på Gitlab (TDP019 Gitlab). Juliet har utvecklats på Ruby version 3.0.2, därmed garanteras det inte att Juliet fungerar på tidigare versioner. Kommandot nedan installerar Ruby på Linux-system.

```
sudo apt install ruby-full
```

2.2 Exekvera din första fil

Olikt kända objekt orienterade programmeringsspråken som Python, C++ eller Java kräver Juliet ingen specifik filändelse endast för Juliet filer, texten/källkoden kan istället skrivas direkt i en txt eller md fil. Juliet är en korsning emellan litteratur och programmering, därför finns det inget krav att användren använder sig utav Juliets nyckelord, däremot finns det inte något att exekvera utan att använda sig utav våra nyckelord som uppgör själva programmeringen i Juliet.

Användaren bör har grundläggande terminal kunskap för att kunna skapa, manövrera och anropa filer. Exekvering av sin första fil sker enligt kommandot nedan, observera att “your_first_file.txt” refererar till användarens egen fil med kod. Sökvägarna till “main.rb” samt egen fil beror på var i filsystemet användaren kör kommandot.

```
ruby main.rb <your_first_file.txt>
```

2.3 Konstruktioner

För att skriva kod i Juliet behöver man självklart veta hur syntaxen ser ut och hur språket används. I delarna 2.3.1 till 2.3.8 förklaras syntax och kodexempel.

2.3.1 Kommentarer = Jargon

Som tidigare nämnts använder Juliet sig av nyckelord endast för att exekvera körbar kod. Därför är allt som inte är reserverade nyckelord eller meningsstrukturer något som kallas för ”Jargon”. Gentemot andra programmeringsspråk, där kommentarer smälter in i koden och kräver någon form av notation för att indikera att det är en kommentar, så smälter koden istället in i texten.

Det är viktigt att notera att man inte kan skriva Jargon efter bindande nyckel ord eller emellan två-nyckelord, gör man det kommer hela uttrycket tolkas som Jargon.

```
    Bellow we're making a simple function.
    the function @fun_function takes @inparameter_x: jargon here is acceptable
        @inparameter_x is @inparameter_x plus 47: then jargon here is unacceptable
        as we then return @x.
```

2.3.2 Datatyper

Juliet är dynamiskt typat vilket innebär att användaren inte behöver specificera typer på sina variabler. Juliet tar in de värden som användaren anger och tar reda på om värdet bör sparas som en integer, float, boolean eller string.

Tilldelning av variabler med hjälp av dynamic typing”kan se ut såhär:

```
@number is 3
@string is "a string"
@boolean_value is true
```

2.3.3 Variabler

Variabler skapas i Juliet med nyckelordet “is” med syntaxen nedan där “name” är namnet på variabeln och “value” är värdet som sparas i variabeln. Namnet på variabeln måste alltid skrivas med “@” i början annars kommer uttrycket tolkas som Jargon.

```
<name> is <value>
```

2.3.4 Operatorer

Operatorer i Juliet skiljer sig från andra programspråk då de skrivs i ord istället för symboler då språket strävar mot att skrivas som flytande text. I Tabell 1 finns Juliets operatorer på vänster sida och dess motsvarighet från matematiken på höger.

Tabell 1: Juliets operander

plus	+
minus	-
multiplied by	*
divided by	/
is smaller than	<
is bigger than	>
is equal to	==
is not equal to	!=

Exempel på hur aritmetiska uttryck kan skrivas med Juliets operatorer.

```
2 plus 4
=> 6
3 minus 2
=> 5
8 divided by 4
=> 2
1 multiplied by 7
=> 7
4 plus 5 multiplied by 2 minus 6
```

Exempel på hur jämförelseoperatorer används i Juliet

```
2 is bigger than 4
=> false
1 is smaller than 2
=> true
8 is equal to 4
=> false
1 is not equal to 7
=> true
```

2.3.5 Kontrollstrukturer

Juliet har ett antal olika konstruktioner för att kontrollera hur och om kod exekveras. While-loopar kör ett kodblock om och om igen så länge som ett angivet sanningsuttryck är sant. For-loopar kör ett kodblock ett specifikt antal gånger som användaren anger. If-satser kör ett kodblock en gång om ett angivet sanningsuttryck är sant.

2.3.5.1 If-satser

If-satser har följande syntax där "expression" är ett sanningsuttryck och "statements" är blocket av koden som ska exekveras om sanningsuttrycket är sant.

```
if the expression <expression> is true, <statements>.
```

En if-sats i Juliet kan se ut så här:

```
if the expression 5 is smaller than 99 is true, 3 plus 2.
```

2.3.5.2 While—loopar

While-loopar har följande syntax där "expression" är ett sanningsuttryck och "statements" är blocket av koden som ska exekveras om sanningsuttrycket är sant.

```
While the expression <expression> is true, <statements>.
```

En while-loop skriven i Juliet kan därmed se ut så här:

```
@x is 0, While the expression @x is smaller than 6 is true, @x is @x plus 1.
```

2.3.5.3 For—loopar

For-loopar har följande syntax där "variable" är en variabel som går igenom alla element i det omfång (range) som angivits. "Range" är ett omfång som antingen kan skrivas som "1 to 10" eller "@ar" där @a är en array och "statements" är blocket av kod som ska exekveras.

```
for <variable> in the range <range>, <statements>.
```

En for-loop i Juliet kan se ut som följande:

```
for @i in the range 0 to 10, @i is smaller than 5.
```

2.3.6 Funktioner

Juliet låter användare skapa egna funktioner som sparar undan block av kod som kan exekveras genom att köra funktionen. Funktioner har följande syntax där "name" är namnet på funktionen och "parameters" är de parametrar som skickas till funktionen. "Statements" är blocket kod som exekveras när funktionen körs och "expression" efter "return" är det värdet som funktionen returnerar (parametrar och "return" är frivilliga och har speciell syntax för när det inte tas med).

```
the function <name> takes <parameters>: <statements>: return <expression>.
<name> taking in: <parameters>.
```

Exempel på funktioner:

```
the function @func takes @y: @y is @y plus 10: return @y.
@func taking in: 5.

the function @noPars takes no parameters: 5 plus 3: return 5.
@noPars taking in: no parameters.

the function @recursive takes @x: @x is @x plus 1, if the expression @x is smaller
    than 9 is true, @recursive taking in: @x...
@recursive taking in: 1.
```

Inbyggda metoder

Juliet har ett antal inbyggda funktioner som anropas med samma syntax som vanliga funktioner. Än så länge finns endast "@print" som skriver ut saker i terminalen och "@input" som låter användaren skriva in ett värde själv.

2.3.7 Array

Juliets arrayer är listor med objekt. Objekten i listan kan vara av vilken datatyp som helst och av olika datatyper. Syntax för att skapa arrayer där "name" är namnet och "values" är de värden som sparas i listan:

```
the array <name> has the values <values>.
the array <name> is empty.
```

Juliets arrayer kan skrivas som följande:

```
the array @cityNames has the values "Linkoping", "Norrkoping", "Stockholm".
the array @randomStuff has the values 3, "hello world", 34.23, "water".
the array @void is empty.
```

2.3.8 Varaibel åtkomst

Det finns vissa regler i Juliet om var man kan använda sina variabler och funktioner på grund av av minneshantering. En variabel eller funktion som skapas inom en funktion, if-sats, for-loop eller while-loop kan inte nås från något annat ställe i koden än blocket den skapades in eller ett yttre block till det block variabeln skapats i. Detta är bra att ha i åtanke då programmet kommer krasha om användaren försöker nå en variabel som inte finns.

Följande kod krashar:

```
if the expression 4 is smaller than 6 is true, @y is 9.
@y plus 3
```

För att undvika denna typ av problem kan användaren skapa en variabel utanför blocket och sedan ändra värdet på den befintliga variabeln istället för att skapa en ny. Då kommer variabeln finnas kvar.

Följande kod krashar inte:

```
@y is 0
if the expression 4 is smaller than 6 is true, @y is 9.
@y plus 3
```

3 Systemdokumentation

Juliet består av fem moduler, RDparse, och juliet.rb för parsning och den lexikaliska analysen, nodes.rb för att skapa ett syntaxträd, runtime.rb för att exekvera noderna som skapats och built_in_functions för att beskriva inbyggda funktioner (print och input).

3.1 Lexikalisk analys och parsning

Modulerna RDparse och Juliet gör den lexikaliska analysen av koden för att skapa tokens som används i en semantisk matchning för att skapa ett AST (Abstract Syntax Tree). Juliet har många långa och specifika nyckelord för att undvika att kod parsas som jargon. När tokens skapas i Juliet matchas först mellanslag och mellanslag mellan citattecken (mellanslag som är del av en sträng). De första ignoreras och de andra returneras. Därefter matchas de nyckelord och tecken som utgör större delen av den kod som Juliet faktiskt exekverar (operatorer, kontrolstrukturer, funktioner osv).

Juliet hanterar tokens för följande:

- Operatorer
- Artimetiska uttryck
- Uttryck för kontrollstrukturer
- Funktionsuttryck
- Strängar
- Jargon

Med de tokens som skapas används reglerna i juliet.rb för att skapa språkets grammatik, grammatiken följer Juliets BNF och uttrycker vilken sammansättning av tokens som skapar respektive konstruktion i Juliet.

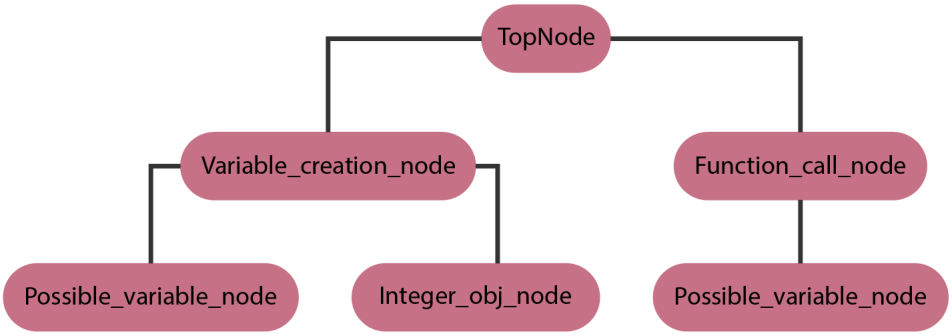
3.2 AST

Juliets konstruktioner skapas genom att matcha tokens till en specifik grammatisk uppbyggnad. Varje konstruktion har sin motsvarande nodklass i filen nodes.rb som används för att skapa de nod-objekt som uppgör nodträdet.

Nodträdet är en lista av noder som användes som inparametrar till topnode-objektet som instansieras i juliet.rb. Eftersom varje nodklass har sin respektive evaluate-funktion kan topnode kalla på sin interna evaluate-funktion som då successivt evaluerar nodträdet genom att kalla på nodernas evaluate-funktion.

Nedan följer ett exempel på en bit kod samt dess respktive AST i Figur 1.

```
1    first we'll create a variable, the @variable is 5
2    then we're going to run the function @print taking in: 5.
```



Figur 1: AST (abstrakt syntax träd).

3.3 Runtime

Juliets Runtime hanteras av filerna runtime.rb, nodes.rb och built_in_functions.rb. Alla noder i AST:t har evalueringsfunktioner som delvis beskriver vad den konstruktionen gör. Vissa noder beskrivs helt i sin egen evaluering och andra har funktioner i runtime.rb som beskriver dem. Funktionerna i runtime.rb är främst för de noder som har delvis liknande beteende men även för de noder som kräver mycket kod. built_in_functions.rb är aktiv i runtime då det endast är under runtime som den kan köras.

3.3.1 Scope

Scope och minneshantering sker i runtime.rb i form av två olika klasser, Stack och Scope. Stack agerar som en lista av alla scopes, instanser av Stack skapas endast i Juliet-klassens initialize-funktion. Stack har tre viktiga funktioner, append, get och pop. Append lägger till ett nytt scope i listan, get hämtar det översta scopet i listan och pop tar bort det översta scopet i listan. Dessa funktioner gör hantering av scope enkelt ock intuitivt. När till exempel en for loop körs anropas append som skapar ett scope åt loopen. Därefter exekveras koden i loopen och när loopen är färdig anropas stack:s pop så att scopet tas bort.

Scope-klassen hanterar individuella scope. Klassen har två hashtabeller som den använder för att separat spara variabler och funktioner där nycklarna är det instansierade variabel- eller funktions-namnet. Dessa Ruby hashtabeller kan hämtas och skrivas över med hjälp av scope-klassens inbyggda get- och set-variable/functions-metoder. Scope-klassen kräver också att den skapas med id och parent_id. parent_id används för att titta igenom yttre scope. En instans av global scope är instansierad i juliet.rb med parent_id nil. Detta används för att rekursivt se om yttre scope innehåller sökt variabel eller funktion tills den når det globala scopet. Om variabeln eller funktionen inte hittats när funktionen når det globala scopet kan slutsatsen dras att den inte finns. Detta rekursiva beteende sker både under sökande och modifierande av variabler och funktioner.

debug_print

För kunna visualisera hur variabler, funktioner och inre scope sparades i gloabalscope behövdes en lätt funktion för att printa ut det i terminalen. Denna funktion är inte tillgänglig för användare men är ett bra verktyg för utvecklare.

3.4 Kodstandard

Juliet tillåter användaren att uttrycka sig fritt igenom text utöver de givna nyckelorden/nyckelmeningarna och därför sätter vi inga specifika kodstandarder för användaren förutom att användaren måste skriva “@” framför sina variabel- och funktionsnamn. Juliet utmanar användaren att vara så beskrivande i sin text som möjligt eftersom det är ett estoresikt språk. Därmed är användningen av mycket Jargon uppmuntrad.

4 Erfarenheter och Reflektion

Nu är projektet i princip färdigt och vi har lärt oss mycket under arbetets gång. Juliet fungerar i stort sett som det ska och vi har uppnått våra mål med språket.

4.1 Planering

TDP019-Kursen hade tydligt stadgade deadlines för när språket och dokumentationen skulle vara klara. Utmaningen blev därför att förstå vilka moduler och konstruktioner som behövde finnas för att nå den nivå av komplexitet vi önskade och därifrån göra en tidsuppskattning för hur lång tid det skulle ta att utveckla.

Vi gjorde milstolpar för varje konstruktion som skulle skapas och delade in projektet i två faser, en för implementation och en för dokumentation. Vi valde att inte specifiera en deadline för varje konstruktion eftersom det var svårt att förutse om vi behövde eller hur mycket vi skulle behöva debugga varje individuellt moment. Däremot hade vi kunnat abstrahera programmet mer och gjort en tidsplan efter lexern, parsern, nod-klasserna och runtime-modulen för att lära oss planera mer.

En erfarenhet att bära med sig till nästa planeringsfas är att anta att utvecklingen för varje konstruktion kommer ta längre tid längre in i utvecklingen man har kommit eftersom den generella komplexiteten har ökat och får per automatik högre coupling.

Vi planerade även att använda enhetstester för varje konstruktion vi skapade. Tyvärr följde vi inte den planeringen och vi fick därför oväntade felfall trots att klasserna kanske betedde sig som förväntat.

4.2 Genomförande

Vi började utvecklingen med att implementera BNFn i parsern så att tokeniseringen matchade den grundläggande grammatiken för Juliet.

Vårt första misstag gjordes efter att vi hade utvecklat Juliets aritmetik. Vi var snabba i genomförandet och med attityden att genomföra och testa för att lära oss. Det vi inte visste gick vi direkt på att utveckla. till exempel klasserna för Juliets kontroll strukurer. Eftersom vi tänkte aritmetik som matematik tänkte vi inte på att booleans, strings och chars kunde tilldelas till en variabel etc och ingen hänsyn togs till det. Det misstaget kostade oss mycket tid i debugging och upptäcktes efter att våra tester för funktioner inte fungerade.

När vi sedan försökte implementera de andra grundläggande datatyperna (float, boolean, string) fungerade inte våra mer komplicerade strukturer längre. Detta gick att lösa men det hade varit lättare om vi hade haft enhetstester att köra när vi implementerade de grundläggade datatyperna för att veta mer exakt när allting slutade fungera. För resten av projektet arbetade vi med enhetstester för alla konstruktioner vi hadde.

Under projektets gång hade vi mycket problem med Jargon. Antingen matchade kod som skulle exekveras som Jargon eller så matchades Jargon som kod som skulle exekveras och programmet krashade. Speciella tecken som matchades som antingen del av exekverbar kod eller som Jargon skapade också många problem. På grund av det är punkter exluderade från Jargon för tillfället. En möjlig uppdatering för Juliet är därmed att byta ut de punkter som används i exekverbar kod mot något nyckel ord så att punkter kan användas fritt i Jargon.

4.3 Lärdomar

Detta projekt har utvecklat hur vi ser på programmeringsspråk och vi har utvecklat förståelsen för hur kod tolkas och exekveras med lexers, parsers och abstrakta syntax-träd. Vi har lärt oss hur man skapar och tolkar BNF:r och samtidigt utvecklat våra kunskaper i ytterligare ett programmeringsspråk, Ruby.

5 BNF

⟨*program*⟩ ::= ⟨*statements*⟩

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

⟨*statement*⟩ ::= ⟨*assignment*⟩ | ⟨*control_structure*⟩ | ⟨*expression*⟩ | ⟨*jargon*⟩

⟨*jargon*⟩ ::= [A-Za-z0-9åäöÅÄÖ!()?]+

⟨*control_structure*⟩ ::= ⟨*for*⟩ | ⟨*while*⟩ | ⟨*if_else*⟩

⟨*for_loop*⟩ ::= "for" ⟨*identifier*⟩ "in the range" ⟨*range*⟩ "," ⟨*statements*⟩ "."

⟨*range*⟩ ::= ⟨*factor*⟩ `to` ⟨*factor*⟩ | ⟨*identifier*⟩

⟨*while*⟩ ::= `while the expression` ⟨*expression*⟩ `is true,` ⟨*statements*⟩ "."

⟨*if_else*⟩ ::= ⟨*if*⟩ | ⟨*if*⟩ ⟨*else_if*⟩* | ⟨*if*⟩ ⟨*else*⟩ | ⟨*if*⟩ ⟨*else_if*⟩* ⟨*else*⟩ "."

⟨*if*⟩ ::= `if the expression` ⟨*expression*⟩ `is true,` ⟨*statements*⟩ "."

⟨*else_if*⟩ ::= `else if` ⟨*expression*⟩ `then,` ⟨*statements*⟩ "."

⟨*else*⟩ ::= `else` ⟨*statements*⟩ "."

⟨*assignment*⟩ ::= ⟨*variable_assignment*⟩ | ⟨*function_assignment*⟩ | ⟨*class_assignment*⟩

⟨*variable_assignment*⟩ ::= ⟨*identifier*⟩ `is` ⟨*expression*⟩ | ⟨*array*⟩ | ⟨*hash*⟩

⟨*function_assignment*⟩ ::= `the function` ⟨*identifier*⟩ `takes` ⟨*parameters*⟩ "and then," ⟨*statements*⟩ "." | `the function` ⟨*identifier*⟩ ⟨*statements*⟩ `.`

⟨*call*⟩ ::= ⟨*function*⟩

⟨*function*⟩ ::= ⟨*identifier*⟩ "(taking in:" ⟨*factors*⟩ ")"

⟨*array*⟩ ::= `the array` ⟨*identifier*⟩ `has the values` ⟨*array_values*⟩ | `the array` ⟨*identifier*⟩ `is empty`

⟨*array_values*⟩ ::= ⟨*value*⟩ | ⟨*value*⟩ `,` ⟨*array_values*⟩

⟨*hash*⟩ ::= `the hash` ⟨*identifier*⟩ `has the key,value pair` ⟨*pairs*⟩ `.` | `the hash` ⟨*identifier*⟩ `is empty`

⟨*pairs*⟩ ::= ⟨*pair*⟩ | ⟨*pair*⟩ `,` ⟨*pairs*⟩

⟨*pair*⟩ ::= ⟨*key*⟩ `:` ⟨*value*⟩

⟨*key*⟩ ::= ⟨*string*⟩

⟨*value*⟩ ::= ⟨*factor*⟩

⟨*expression*⟩ ::= ⟨*call*⟩ | ⟨*factor*⟩ | ⟨*expression*⟩ ⟨*op*⟩ ⟨*factor*⟩ ⟨*factors*⟩ ::= ⟨*factor*⟩ | ⟨*factor*⟩ ⟨*factors*⟩ ⟨*factor*⟩ ::= ⟨*number*⟩ | ⟨*string*⟩ | ⟨*variable*⟩ | ⟨*boolean*⟩ | `(` ⟨*expression*⟩ `)` | ⟨*unary_op*⟩ ⟨*factor*⟩

⟨*op*⟩ ::= ⟨*binary_op*⟩ | ⟨*bool_op*⟩

⟨*bool_op*⟩ ::= `is smaller than` | `is larger than` | `is equal to` | `is smaller than or equal to` | `is larger than or equal to`

⟨*binary_op*⟩ ::= ⟨*unary_op*⟩ | ⟨*mult_op*⟩ | ⟨*div_op*⟩

⟨*parameters*⟩ ::= ⟨*parameter*⟩ | ⟨*parameter*⟩ `,` ⟨*parameters*⟩ | `no parameters`

⟨*parameter*⟩ ::= [A-z] och siffror

⟨*identifier*⟩ ::= ⟨*string*⟩

⟨*variable*⟩ ::= ⟨*string*⟩

⟨*unary_op*⟩ ::= `plus` | `minus`

⟨*mult_op*⟩ ::= `multiplied by` | ⟨*div_op*⟩

⟨*div_op*⟩ ::= `divided by`

⟨*boolean*⟩ ::= `0` | `1` | `true` | `false`

⟨*string*⟩ ::= ⟨*char*⟩+

⟨*number*⟩ ::= ⟨*integer*⟩ | ⟨*float*⟩

⟨*integer*⟩ ::= ⟨*digit*⟩+

⟨*float*⟩ ::= ⟨*digit*⟩+ `.` ⟨*digit*⟩*

⟨*digit*⟩ ::= [0-9]

⟨*char*⟩ ::= [A-z]