SPRAK

TDP019 Projekt: Datorspråk

Andreas Magnvall och Simon Murhed

Institutionen för datavetenskap

Linköpings universitet

VT – 2019

2019-05-14

Table of Contents

Inledning

Detta är ett projekt gjort av Andreas Magnvall och Simon Murhed i kursen TDP019 – Projekt: Datorspråk som hölls vårterminen 2019. Projektet gick ut på att skapa ett eget programmeringsspråk, med hjälp av rdparse. Slutresultatet valde vi att kalla Sprak och i detta dokument kommer vi att gå igenom hur språket är uppbyggt samt hur det används.

Syfte

Syftet har varit att skapa ett programmeringsspråk med inspiration av C-språkssyntaxen och Python. Exempelvis är måsvingar och vissa nyckelord tagna från C-språken samtidigt som ett mindre inslag av parenteser är tagna från Python. Målet har varit att skapa ett språk för betygsgränsen 3 och om tid skulle finnas skulle även försök gjorts för att uppnå betygsgräns 4.

Resultat

Projektet resulterade i ett programmeringsspråk där syftet för betygsgräns 3 var uppnådd. Sprak stöder grundläggande programering som en annan programmerare kan känna igen sig i.

Målgrupp

Sprak är inte direkt anpassat för en specifik målgrupp. Under projektets gång ändrades mycket av syntaxen och uppbyggnaden och slutresultatet blev en mix av C++ och Python. Språket i slutändan visade sig vara anpassat för de som gillar C++ syntax men är trötta på all typning som alltid krånglar.

Användarhandledning

Installation

För att köra Spraks parser krävs version 2.5 av Ruby.

Sprak körs i nuläget endast genom filinläsning. För att köra en fil lokaliserad i samma mapp som parsern själv går det att skriva “ruby sprak.rb” följt av “*./filnamn*”, *t.ex. “ruby sprak.rb ./testfil”* i terminalen.

Datatyper

Sprak är dynamiskt typat vilket innebär att typer inte behöver specificeras. Logiskt i bakgrunden finns det dock 3 datatyper, booleska värden, heltal och strängar. Dessa matchas mot respektive regexuttryck. I nuvarande version av sprak kan strängar ej inkludera mellanrum, istället kan understreck användas.

boolean	string	digit
'true'	/"/, /.+/, /"/)	/\d+/
'false'		

Tilldelning och variabler

Variabler tilldelas med tilldelningsoperatorn =. Sprak stöder inte fördefinierade icke instansierade variabler utan detta sker vid tilldelning.

Logiska operatorer

Sprak har tre grundläggande logiska operatorer. Likt de flesta programmeringsspråk kan sprak hantera and-, or-, samt not-operatorer. And-operatorn kräver att båda uttrycken evalueras till sant, i or-operatorn räcker det däremot att endast ett uttryck blir sant. Not-operatorn används för att få ut motsatsen till ett logiskt uttryck. För att använda dessa operatorer skrivs “*booleskt uttryck and booleskt uttryck*” för and, på samma sätt för or, och “*not booleskt uttryck*” för inversen av ett uttryck.

Uttryck:	**Evalueras till:**
true and false	false
true or false	true
not true	false

Jämförelseoperatorer

Sprak har följande operatorer för jämförelser av två värden:

Lika med	Mindre eller lika med	Större eller lika med	Mindre än	Större än	Inte lika med
==	<=	>=	<	>	!=

Dessa jämförelser sker efter matematiska operatorer men innan logiska operatorer och de returnerar ett booleskt värde

Matematiska operatorer

Sprak kan hantera de vanligaste operatorerna inom matematiken, addition, subtraktion, multiplikation samt division. Ordningen av dessa operatorer sker även i korrekt ordning, dvs. Multiplikation och division sker före addition och subtraktion. Sprak stöder i nuvarande form inte parenteser vid beräkning men stöd för detta kan enkelt läggas till. Matematiska operatorer är de första operationerna som sker då programkoden parsas.

Villkorssatser

Sprak stöder villkorssatser i form av if, elif och else. Logiskt går de bara att skriva i ordning med ett flertal elif. Så fort ett av dessa villkor blir uppfyllt och parsern går in i en av if-satserna kommer resterande kodblock inte evalueras. I if och elif måste måsvingarna innehålla minst en rad kod för att parsas och evalueras på ett korrekt sätt.

```
if false
{
        println “körs_ej”
}
elif false
{
        println “körs_ej”
}
elif true
{
        println “körs!”
}
else
{
        println “körs_ej”
}
```

Repetitioner

Sprak har två traditionella loopar, for-loopar och while-loopar. Precis som i de flesta programmeringsspråken fungerar dessa loopar på olika sätt. For-loopen itererar över ett stycke kod ett förbestämt antal gånger medans while-loopen fortsätter iterera över koden till ett visst krav är uppfyllt.

For-loopar

For-loopar i Sprak skrivs med 3 argument. En variabeltilldelning, ett villkor och ytterligare en variabeltilldelning där den egna variabeln är tänkt. Den kod som ska köras varje varv skrivs inom måsvingar.

While-loopar

While-loopar i Sprak skrivs bara med ett argument, ett villkor. Loopen körs tills detta villkor har uppfyllts. På samma sätt som med for-loopar skrivs den kod som ska köras innanför måsvingar.

```
//for-loop; skriver ut 1 till 10
for i = 0; i <= 10; i = i+1
{
        println i
{

//while-loop; skriver ut 2, 4, 8, 16, 32
j= 2
while j < 20
{
        print j
        j = j*2
}
```

Funktioner

Funktioner i sprak kan skrivas med eller utan parametrar. Likt andra språk måste lika många argument skickas med som specificerats av antal parametrar. I nuvarande form finns ej standardvärden men språket kan relativt enkelt utökas med det. Funktioner har ingen begränsning av antal parametrar. Funktioner kan bestå av små och stora bokstäver och måste vara minst en bokstav långt. Funktioner definieras med nyckelordet func i början, dvs “func *funktionsnamn*(*variabelnamn*) { *uttryck* }. Funktioner anropas genom funktionsnamnet och parametrarna; *funktionsnamn*(*parametrar*)

Nedan visas ett exempel på funktionsdefinition samt funktionsanrop.

```
        func double(number)
        {
                return number * 2
        }

        twenty = double(10)
```

Scope

Sprak stöder scopehantering. Exempelvis inkluderade detta lokala variabler. Ett nytt scope öppnas vid funktionsanrop och variabler deklarerade i detta scope tas bort då funktionen når sitt slut. Då ett scope öppnas kopieras det tidigare scopet över för åtkomst av dessa variabler. Även dessa kopior försvinner då funktionen har körts färdigt. I nuläget används scope endast vid funktionsanrop.

Utskrift

Värden kan skrivas ut med funktionerna print och println, print skriver ut ett värde eller sträng utan radbyte medans println har ett radbyte på slutet.

Kommentarer

Kommentarer i Sprak skrivs med // i början. Dessa kommentarer gäller för resten av raden de är skrivna på.

Systemdokumentation

Lexikalisk analys

Den lexikaliska delen av parsningen består av att matcha det vi söker och skapa (tokens). Detta sker med regex. Matchningen av tokens är designad att vara relativt generell och matcha all indata med endast ett fåtal regex-uttryck.

Parsing

Parsningen sker med de BNF-regler som bestämt i vår BNF. Först jämförs den egna koden mot våra regler och utifrån vad som matchas skapas olika objekt. Den övergripande regeln är ett objekt som motsvarar en lista av statements och/eller enbart ett statement. Denna lista kan beskrivas som alla operationer programmet består av. Ett statement kan exempelvis vara deklaration och instantiering av variabler, skapande av funktioner och kallelse på funktioner. Detta blir en sorts trädstruktur där alla regler har olika prioriteter. När parsning sker så sker det rekursivt.

Evaluering

Alla de objekt som skapas vid parsningen har en funktion för evaluering. Det är dessa funktioner som i tur och ordning exekverar de kodstycken som byggts upp i parser-trädet då koden parsades. Programmet kommer fortsätta att gå ner i alla grenar i detta träd tills slutet har nåtts och all kod har evaluerats.

Reflektion

Det var från början väldigt svårt att komma ihåg och förstå vad som skulle göras. Det var också svårt att förstå hur vi skulle använda det som tagits upp på föreläsningar och till viss mån det vi lärt oss från TDP007. Visst finns det likheter men mycket var helt nytt och ett helt annat tänk. Ofta kan projektet ses som att det enbart finns en lösning på hur det ska utföras. Ett exempel är att objekten som skapas av parsningen har en eval metod. Detta var ingen vi tänkt på innan eller känt att vi lärt oss av det vi gått igenom. Det är svårt att veta hur vi skulle kommit på detta på egen hand. Visst kan man tänka att mer tid kunde lagts på projektet i början men samtidigt går det att fråga sig hur enkelt det är när man inte lyckas komma igång. Om något gick bra var det kanske att vi enkelt fick till stöd för aritmetiska uttryck tidigt.

Efter att vi lärt oss om den egna evalfunktionen gick det enklare. Snabbt kunde vi skapa klasser för egna objekt i språket och olika operationer. Till viss del fanns det flertal motgångar där någon rad haft väldigt stor påverkan för att lösa problemet. Det svåra var inte att veta vad som skulle göras längre och inte heller hur. Istället var det svåra att lösa problemen som uppstod med hur vi tänkt. Kanske hade det varit bra idé att vara ännu bättre på både Ruby och tänka över hur vi tänkt kring implementationerna innan.

I slutet gick mycket tid till att fixa fel. Vi försökte även fixa rekursion men lyckades inte fullt ut. Tester lades också till sent. Dessa kunde lagts till tidigare för ibland uppstod det problem med något helt annat än det man höll på med. Samtidigt var det svårt att testa något vi inte kommit långt med.

BNF

```
<program>    ::= <stmt_list>

<stmt_list>    ::= <stmt><stmt_list>
                   | <stmt>

<stmt>          ::= <io_stmt>
                   | <return_stmt>
                   | <condition_stmt>
                   | <iter_stmt>
                   | <func_stmt>
                   | <assign_stmt>
                   | <expr>


<io_stmt>      ::= <print_stmt>

<print_stmt> ::= 'print'   <expr>
                   | 'println' <expr>



<iter_stmt>   ::= <for_stmt>
                   | <while_stmt>

<for_stmt>     ::= 'for' <assign_stmt> ';' <or_expr> ';' <assign_stmt> ';' '{' <stmt_list> '}'

<while_stmt> ::= 'while' <or_expr> '{' <stmt_list> '}'



<func_stmt>  ::= <func_def>

<func_def>   ::= 'func' <func_id> '(' <params> ')' '{' <stmt_list> '}'
                   | 'func' <func_id> '('')' '{' <stmt_list> '}'

<params>     ::= <param>
                  | <params> <param>

<param>      ::= /[A-Za-z]+/

<func_id>    ::= /[A-Za-z]+/

<return_stmt>::= 'return' <expr>
```

```
<func_call>  ::= <func_id> '(' <args> ')'
               |  <func_id> '(' ')'

<args>       ::= <arg>
               |  <args> <arg>

<arg>        ::= <expr>



<assign_stmt>::= <var> '=' <expr>


<condition_stmt>::= <if_stmt>
                  | <elif_stmt>
                  | <else_stmt>

<if_stmt>    ::= 'if' <expr> '{' <stmt_list> '}'
<elif_stmt>  ::= 'elif' <expr> '{' <stmt_list> '}'
<else_stmt>  ::= 'else' '{' '}'
               |  'else' '{' <stmt_list> '}'


<expr>       ::= <or_expr>

<or_expr>    ::= <or_expr> 'or' <and_expr>
               |  <and_expr>

<and_expr>   ::= <and_expr> 'or' <and_expr>
               |  <not_expr>

<not_expr>   ::= 'not' <not_expr>
               |  <comp_expr>


<comp_expr>  ::= <add_expr> <logic_comp> <add_expr>
               |  <add_expr>

<logic_comp> ::= '=='
               |  '<='
               |  '>='
               |  '>'
               |  '<'
               |  '!='


<add_expr>  ::= <add_expr> '+' <mult_expr>
              |  <add_expr> '-' <mult_expr>
```

```
                  |  <mult_expr>

<mult_expr> ::= <mult_expr> '*' <mult_expr>
                  |  <mult_expr> '/' <mult_expr>
                  |  <variables>


<variables> ::= <func_call>
                |  <digit>
                |  <string>
                |  <boolean>
                |  <var>

<digit>       ::= /\d+/

<string>      ::= '"' /.+/ '"'

<boolean>   ::= 'true'
                 |  'false'

<var>      ::= /[A-Za-z]+/
```