TDP019 Projekt: Datorspråk

Nyan

Författare

Hemming Niku, hemni276@student.liu.se
Julia Mak, julma873@liu.se

Vårterminen 2024

Linköpings universitet 2024-05-15

Contents

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Första versionen skapad	2024-03-26
1.1	Första utkast	2024-05-07
1.2	Uppdaterat BNF-grammatiken	2024-05-13

2 Inledning

Under andra terminen på programmet Innovativ Programmering studerades kursen TDP019 Projekt: datorspråk där målet är att konstruera ett eget, enklare programmeringsspråk. Inspirationen för projektet var att skapa ett imperativt språk som inspireras av katter och ljuden de gör.

3 Användarhandledning

Som tidigare nämnts, inspireras koden av katter och ljuden de gör vilket gör Nyans syntax unik. Därför anses målgruppen vara andra programmeringsstudenter eller personer med en viss programmerings erfarenhet. Nyan riktar sig också mot potentiella arbetsgivare då projektet kan påvisa hur ett mindre datorspråk kan implementerats.

Inom språket finns många fundamentala konstruktioner som funktioner, while-loopar, aritmetik och if-satser. Dessa konstruktioner ger, i kombinationer med varandra, möjligheten att skapa enkla program i likhet med fizz-buzz eller enkla räknare.

I exemplet fizz-buzz beskrivs programmet som en räknare denna räknare byter ut vissa siffror med ordet fizz, buzz eller fizz-buzz. Fizz ersätter siffror delbara med tre, buzz ersätter siffror delbara med fem och fizz-buzz ersätter siffror delbara med både fem och tre.

3.1 Installation

För att att kunna använda Nyan går det till exempel att klona ner projektet från Gitlab eller extrahera filerna från en tar-fil.
Om projektet är paketerat som en tar-fil går det att extrahera filerna på följande sätt:

```
tar zxvf nyan.tar.
```

För att hämta projektet från gitlab, öppna en terminal och skriv in följande kommando:

```
git clone https://gitlab.liu.se/julma873/tdp019
```

3.2 Ruby version

För detta projekt har Ruby version 2.7.4 används. Installering av Ruby i terminalen:

```
sudo apt-get install ruby
```

3.3 Interpretator

Nyan kan köras i antingen interaktivt läge eller läsa in en fil.

För att få hjälp, köra programmet med debug-mode eller se vilken version av Nyan man kör, går det att använda olika flaggor. Se exempel nedan samt tabell 1 för mer information om vad flaggorna står för.

För att skriva i interaktivt läge:

```
ruby nyan.rb
```

Nedanför beskrivs syntaxen för hur det går att köra programmet med filer:

```
ruby nyan.rb [filnamn]
```

För att visa eller köra med olika alternativ:

```
ruby nyan.rb [filnamn] | [flagga]
```

Flagga	information
-h, –help	Display help information
-d, –debug	Set debug mode true or false
-v, –version	Display the latest version
<filename>	Read from script file

Table 1: Kommandoradsargument

3.4 Syntax

Som tidigare nämnts är Nyans syntax baserat på katter och dess ljud de gör. För att tydliggöra går det att se i tabell 2.

Något som skiljer sig från till exempel Python och Ruby är att det finns ett start- och slut tecken för ett block. När ett block definieras i en while-loop, nästlade if-satser eller en function, öppnar man med ':' och avslutar med ':3'. Istället för parenteser används '^' '^'. Nedan finns kod-exempel skrivet i Nyan.

nyan-syntax	definition
?nya?	if
?nyanye?	else-if
?nye?	else
meow	print
prrr	while
mao	def
hsss	return

Table 2: Nyan-syntax

3.4.1 Datatyper

Nyan använder sig av statisk typning för att deklarera olika typer av variabler. Dessa olika typer representeras genom olika kattansikten. Se tabell 3.

^w^	sträng
^3^	integer
^.^	float
^oo^	boolean

Table 3: Datatyper

3.4.2 Variabler

Nedan finns ett kodexempel för hur olika variabler tilldelas ett värde:

```
^w^ string = "Defines a string"~ => string
^3^ x = 42~
^.^ y = 3.14~
^oo^ boolean = true~
```

Något som är viktigt att notera är att tilldelningen slutar med ett tilde tecken '~'

3.4.3 If-statser

```
?nya? ^3 < 1^:
    meow ^"false"^
?nyanye? ^1 < 3^:
    meow ^"?nyanye? stands for elseif"^
?nye?:
    meow ^"?nye? equals else"^
:3
```

3.4.4 Repeitionssats

```
^3^ x = 0~
prrr ^x < 5^:
    ?nya? ^true^:
        meow ^"meowed"^
    :3
    x+=1~
:3
```

3.4.5 Array

För att skapa en array med heltal:

```
^3^ arr=[]           => []
^3^ arr =[1, 2, 3]~  => [1, 2, 3]
```

Åtkomst till element :

```
^3^ arr =[1, 2, 3]~
arr[0] => 1
```

Ta bort ett element :

```
^3^ arr =[1, 2, 3]~
arr.pop => 3
```

Lägg till ett element :

```
^3^ arr =[1, 2, 3]~
arr.push(4) => arr[1, 2, 3, 4]
```

Storlek av en array :

```
^3^ arr =[1, 2, 3]~
arr.size => 3
```

3.4.6 Funktioner

I följande exempel definieras en funktion utan en in-parameter och inuti finns en if-sats som skriver ut strängen 'hello'.

```
mao test^^:
    ?nya? ^2 < 5^:
        meow ^"hello"^
    :3

    meow^y^
    hsss y

    meow^"dont print this"^
:3
meow ^ test^^ ^ => hello\n 42
```

Rekursion:

```
mao recurs^x^:
?nya? ^x < 3^:
    meow ^x^
    recurs^x + 1^
    :3
:3
^3^ z = 0~
recurs^z^
```

4 Systemdokumentation

Nyan är ett imperativt programmeringsspråk som använder sig av statisk typning och dynamisk scopehantering. Språket kombinerar funktionalitet från både Python och Ruby. För att genomföra den lexikaliska analysen och parsningen användes RDparse. Därefter genereras ett abstrakt syntaxträd med noder. Dessa noder är objekt av olika klasser för olika konstruktioner.

4.1 Lexikalisk analys

För den lexikaliska analysen av språket används den givna koden i filen RDparse. RDparse läser in alla tokens som skapas först och använder sedan dessa tokens i ordningen som beskrivs i de unika match-casen.

Nedan visar tokens som genereras och finns i filen *parser.rb*.

```
token(/\s+/)
token(/\:3/)            {|_| ';'}
token(/[\d]+\.[\d]+/)   {|m| m}
token(/\d+/)            {|m| m }
token(/".*"/)           {|m| m}
token(/\^w\^/)          { :string }
token(/\^3\^/)          { :integer }
token(/\^\.\^/)         { :float }
token(/\^oo\^/)         { :boolean }
token(/\bprrr\b/)       { :whileloop }
token(/\^/)             {|m| m}
token(/\(/)             {|m| m}
token(/\)/)             {|m| m}
token(/mao/)            { :def }
token(/meow/)           { :meow }
token(/push/)           { :push }
token(/pop/)            { :pop }
token(/size/)           { :size }
token(/\?nya\?/)        { :if }
token(/\?nye\?/)        { :else }
token(/\?nyanye\?/)     {:elseif}
token(/[[a-zA-Z]\d_]+/) {|m| m}
token(/,/)              {|m| m}
token(/\./)             {|m| m}
token(/\[/)             {|m| m}
token(/\-\=|\+\=|\+|\-|\*|\&\&|\|\||\=\=|\/\/|\%|\<|\>|\=|\]|\~|\:/) {|m| m}
token(/./)              {|m| m}
```

Reguljära uttryck används i alla tokens för att skilja på tecken, variabler och symboler. Alla tecken tokens används för att separera tecken från variabler och objekt så att deras funktion definieras väl, exempelvis separeras '=' från '=='. Variabler har ett definierat regex-uttryck som har en mer flexibel matchning så att hela variabelnamn kan returnerats med siffror, bokstäver och binde- samt understreck inkluderade.

Nyan använder sig av symboler som genereras av ord man skriver i språket. Exempel på dessa är 'meow' som genererar ':print', '?nya?' som genererar ':if' och 'mao' som genererar ':def'. Dessa används för att särskilja specifika beteenden genom inbyggda funktioner som funktionsdefinitioner, while-loopar och terminal-utskrifter. För att använda lexikalisk analys effektivt behövs dessa för att tydligt visa var visst beteende börjar och slutar.

4.2 Parsning

Utifrån den lexikaliska analysen parsars koden enligt den givna grammatiken nedan.

BNF grammatik:

```
<program> ::= <component>

<component> ::= <blocks>

<blocks> ::= <block> | <blocks> <block>

<block> ::= <while> | <while> | <condition> | <arrayOp> | <return> | <functionCall> | <function>
                    | <arrayOp> | <reassignment> | <assignment> | <print> | <expr>

<assignment> ::= <datatype> <variable> "=" <output> "~"

<reassignment> ::= <variable> "+=" <value> "~"
                   | <variable> "-=" <value> "~"
                   | <variable> "=" <output> "~"

<print> ::= :meow "^" <value> "^"
            | :meow "^" <output> "^"

<output> ::= <arrayOp> | <array> | <functionCall> | <variable> | <expr>

<array> ::= "[" <elements> "]"

<elements> ::= <value> "," <elements> | <value>

<arrayOp> ::= <variable> "[" <value> "]"
              | <variable> "." :push "^" <value> "^"
              | <variable> "." :pop
              | <variable> "." :size

<values> ::= <value> | <variable>

<function> ::= :def <variable> "^" <params> "^" ":" <blocks> ";"
               | :def <variable> "^" "^" ":" <blocks ";"

<functionCall> ::= <variable> "^" <params> "^"
                   | <variable> "^" "^"

<params> ::= <expr> | <variable> | <params> "," <variable>

<condition> ::= "if" "^" <logicStmt> "^" ":" <blocks> ";" <conditionFollowup>
                | "if" "^" <logicStmt> "^" ":" <blocks> ";"

<conditionFollowup> ::= "else" ":" <blocks>
                        | "elseif" "^" <logicStmt> "^" ":" <blocks> <conditionFollowup>
                        | "elseif" "^" <logicStmt>"^" ":" <blocks>

<while> ::= :whileloop "^" <logicStmt> "^" ":" <blocks> ";"

<logicStmt> ::= "not" <logicStmt>
                | <logicStmt> "and" <logicStmt>
                | <logicStmt> "&&" <logicStmt>
                | <logicStmt> "or" <logicStmt>
                | <logicStmt> "||" <logicStmt>
                | <valueComp>

<valueComp> ::= <logicExpr> "<" <logicExpr>
```

```
                | <logicExpr> ">" <logicExpr>
                | <logicExpr> "<=" <logicExpr>
                | <logicExpr> ">=" <logicExpr>
                | <logicExpr> "==" <logicExpr>
                | <logicExpr> "!=" <logicExpr>
                | <logicExpr>

<logicExpr> ::= <value> | <variable> | "(" <logicStmt> ")"

<expr> ::= <expr> "+" <term> | <expr> "-" <term> | <term>

<term> ::= <term> "%" <factor> | <term> "//" <factor> | <term> "*" <factor> | <term> "/" <factor>

<factor> ::= <float> | <int> | <variable> | "(" <expr> ")"

<output> ::= <value> | <variable>

<datatype>  ::= "string"
              | "integer"
              | "float"
              | "boolean"

<variable> [[:alpha:]\d\_]+

<value>  ::= < float>  | < int>  | < bool>  | < string>

<strValue>  ::= ".+"

<intValue>  ::= \d+

<floatValue>  ::= [\d]+\.[\d]+

<boolValue>  ::= "true" | "false"
```

4.3 Klasser och syntaxträd

Vid parsningen och efter att tokens har matchat, skapas olika noder beroende på värdet. Varje nod representeras som en klass och i varje klass finns en eval funktion. När eval körs på den översta noden evaluerar den de underliggande noderna. Nedan finns en lisa över de olika noderna:

- **GlobalScope**
- **Scope**
- **SyntaxTreeNode**
- **ProgramNode**
- **BlocksNode**
- **AssignmentNode**
- **ReassignmentNode**
- **DatatypeNode**
- **VariableNode**
- **ValueNode**
- **PrintNode**
- **ArithmaticNode**
- **ArrayNode**
- **ArrayOpNode**
- **ParamsNode**
- **FunctionNode**
- **FunctionCall**
- **ReturnNode**
- **ConditionNode**
- **LogicStmt**
- **ValueComp**
- **LogicExpr**
- **WhileNode**

4.4 Scope-hantering

Scope-hanteringen består av två delar, GlobalScope och Scope. GlobalScope är det specifika scopet som är ytterst i programmet och har speciell funktionalitet. Den har som uppgift att hålla uppsikt över vilket scope som är aktuellt, samt att den kan skapa eller ta bort scope.

Klassen Scope behöver veta vilka variabler/funktioner som är lokala för den, om det finns ett till scope i sig och vilket dess tidigare scope är. Utöver det har den funktioner för att ta reda på vilket det aktuella scopet är (ifall ett nytt scope skapas), lägga till nya scope, ta bort det aktuella scopet, lägga till variabler och hitta variabler. Många av dessa funktioner ropar bara tillbaka samma funktion in i previous scope eftersom det yttersta scopet har möjlighet att göra mer.

Klassen GlobalScope ärver av Scope och hanterar det mesta. Den vet vilket det nuvarande scopet är och eftersom den har koll på det aktuella scopet så hanterar den tillägg och borttagning också. Eftersom GlobalScope alltid är det yttersta scopet så finns ingen previousScope variabel, däremot finns currentScope för att visa vilket scope som är aktuellt.

Den vet vilket det nuvarande samt aktuella scopet är. Den hanterar även tillägg och borttagning av scope. Eftersom GlobalScope alltid är det yttersta scopet så finns ingen previousScope variabel, däremot finns currentScope för att visa vilket scope som är aktuellt.

Med dessa klasser byggs ett system upp som utgår ifrån det yttersta scopet. För minneshanteringsskäl försvinner scope när funktionen currToPrevScope anropas, funktionens uppgift är att den flyttar currentScope variabeln från det aktuella scopet till det tidigare scopet. Eftersom det varken finns skäl eller möjlighet att återgå till ett tidigare scope så tas det tidigare aktuella scopet bort.

4.5 Kodstandard

Preferensen var att använda sig av camel-case för namngivning i koden. I test cases används snake-case anrop då test modulen är skriven med detta stilval. Namngivningen har målet att vara kortfattade i vad klasser, funktioner och variabler gör. Detta har varierande resultat. Under projektets gång har koden varit konsekvent enligt dessa standarder till en så stor utsträckning som möjligt.

4.6 Filstruktur

Filstrukturen ser ut på följande sätt:

```
TDP019
├── doc
│   └── systemdokumentation_Nyan_julma873_hemni276.pdf
├── nyanTest
│   ├── testFiles
│   │   ├── arithmatics.nyan
│   │   ├── arrays.nyan
│   │   ├── conditions.nyan
│   │   ├── functions.nyan
│   │   └── loop.nyan
│   ├── testConditions.rb
│   ├── testFunctions.rb
│   ├── testLoop.rb
│   ├── testScope.rb
│   └── testSyntaxtree.rb
├── condition.rb
├── demo.nyan
├── errors.rb
├── modules.rb
├── nyan.rb
├── parser.rb
├── rdparse.rb
├── README.md
├── scope.rb
└── syntaxtree.rb
```

5 Programkod

Nyan ligger i en publik utvecklarkatalog på Gitlab och går att se här: https://gitlab.liu.se/julma873/tdp019

6 Erfarenheter och reflektion

Under början av projektet var det förvånande hur svårt det var att påbörja utvecklingen av språket då det inte fanns någon tydlig start-punkt.

Vi hade inte någon bra överblick på hur vi skulle använda RDparse på ett sammanhängande sätt för ett större program. Under början av projektet visste vi inte heller hur programmets struktur skulle se ut.

Efter att ha skrivit bnf-grammatiken noterade vi att det var bra att implementera en del i taget för att kunna få en överblick över projektet. Under projektets gång utformades en lämplig filstruktur. Koden utformades efter att ha utfört många felsökningar vilket ledde till att vi kunna parsa kod. Något som hjälpte oss under utvecklingen var att skriva ett flertal tester, speciellt för de olika klasserna. I testerna gick det också att se hur det parsade men det gav också oförutsedda och oförväntade resultat.

Under utvecklingens gång uppstod problem som gjorde att vi behövde designa om stora delar av programmet. Exempelvis behövde vi tänka om hur vi byggde upp programmet så att det använde sig av objekt i syntaxträd. Stora delar av problemen var baserade på att vi inte hade någon klarhet hur de olika delarna av programmet skulle interagera med varandra. Vilket ledde till att scope-hanteringen samt regler och matchningar fick omarbetas under projektets gång.

Det största hindret vi stötte på under projektets gång var parser-filen. Vi har haft betydligt mindre problem med hur objekten samarbetar än med hur olika delar av språket parsar, något som förmodligen hade underlättats med bättre prioritering av både tokens och match-case.

Det som fungerade bäst var evalueringen för alla noder. De var relativt lätta att implementera. Implementationen av klassen ProgramNode gjorde så att vi bara behövde kalla på eval en gång. Eftersom eval-funktionen bara behövde kallas en gång blev koden lättare att implementera. Ett standardiserat beteende kunde då definieras för de klasser vi skapade i programmet.

I helhet är vi nöjda med slutresultatet av projektet. Det har varit både lärorik och utvecklande. Utöver att utveckla ett programmeringsspråk har vi också blivit mer erfarna i objektorienterad utveckling i Ruby, samt att använda sig av en parser.