LINKÖPINGS UNIVERSITET

TDP019 Projekt: Datorspråk

Systemdokumentation

Författare

Daniel Kaller, danka340@liu.se
Addi Sabanovic, addsa705@liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2023
Version 1.1

2023-05-09

1 Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.1	Färdigställde Användarhandledning, började skriva under rubriken Översiktlig bild av systemet, började skriva på rubriken Klasser för element/objekt i språket under rubriken till Översiktlig bild av systemet	230503
1.0	Första verision av Systemdokumentationen, Implementerade text under rubriken introduktion och började skriva under Användarhandledning	230502

2 Introduktion

”TDP019: Datorspråk” är en projektkurs på IP-programmet som man läser i andra terminen av det första läsåret, där man i en grupp ska konstruera ett datorspråk. I kursen lär man sig bland annat använda sig av lexers och parsers, för att läsa in texten som tokens och därefter skapa syntaxträd utav dem, samt hur man skriver BNF-grammatik för att definiera syntaxen.

Vår grupp valde att skapa ett språk som är en blandning av Python och C++ i semantik och syntax. Vårt språk är imperativt och stödjer de flesta grundläggande funktionaliteterna som förväntas av ett programmeringsspråk, dvs. styrstrukturer, funktioner, etc. Vi har även stöd för ett par inbyggda funktioner, så som print() och split() (den sistnämnda fungerar enbart på strängar i dagsläget).

Vi valde just Python och C++ som våra inspirationer då vi sedan länge var bekanta med de språken, vilket förenklade bestämandet av syntax och semantik. Det gjorde det även lättare för oss att jämföra hur kod i vårt språk fungerar gentemot samma kod fast i Python/C++.

3 Användarhandledning

Detta språk är skapat att ha en blandning av Pythons enkelhet och C++ tydlighet. Exempelvis så är den inbyggda funktionen för utskrift till terminalen i vårt språk ”print()” vilket är hur det skrivs i Python, och är enklare att skriva än ”std::cout «” som man gör i C++.

På annat håll är vårt språk tydligare än Python, då vi till exempel valde att representera kodblock som kod inuti ett par måsvingar ({ och }) istället för kod som finns på samma indenteringsnivå. Vi valde även att använda oss av hård typning, så som i C++, för att undvika otydligheter om vad en variabel ska innehålla.

Något som inte förekommer i varken Python eller C++ är användning av nyckelordet ”end” för att signifiera avslutet på en funktion. Detta tog vi från Ruby, och var något vi gjorde för att göra det enklare att se vart till exempel ett kodblock slutar och en funktion slutar (ett problem vi tyckte förekom för ofta i C++-kursen)

Några exempel på kod i vårt språk är:

```
def factorial(int x)
  if(x == 1) {
    return 1;
  }

  return x * factorial(x - 1);
end
```

I detta exempel definieras en funktion ”factorial” med heltalparametern ”x”. Inuti funktionskroppen finns det en if-sats som kollar ifall x är lika med 1, i vilket fall funktionen returnerar 1. Annars så returnerar funktionen x multiplicerat med ett rekursivt anrop till funktionen med parametervärdet x minus 1.

```
def arithmeticsum(int n, int d)
  int sum = 0;

  for(int i = 0; i <= n; i++){
      sum = sum + i*d;
  }

  return sum;
end

print(arithmeticsum(5, 1));
```

I detta exempel definieras en funktion ”arithmeticsum” med heltalparametrarna ”n” och ”d”. Inuti funktionen finns det en heltalsvariabel ”sum” som sätts till 0. Därefter finns det en for-loop, där kontrollvariabeln ”i” definieras och sätts till 0. Efter det första semikolonet står det att loop:en ska fortsätta så länge som i är mindre eller lika med n, och slutligen så står det att i skall inkrementeras med 1 för varje iteration.

Inuti loop:en så står det att man skall addera sum + i*d till variabeln sum för varje iteration. I slutet av funktionen så returneras sum. Efter funktionskroppen finns det ett anrop till ”print()” där arithmeticsum kallas med parametervärdena 5 och 2. Detta ger en utskrift av det slutgiltiga returvärdet av funktionen till terminalen.

De konstruktioner som finns i språket är:

- grundläggande datatyper (int, float, bool, string, array)
- variabler
- aritmetiska och logiska uttryck
- if-satser och else if- samt else-satser
- for- och while-loopar
- funktioner
- rekursiva funktionsanrop

4 Översiktlig bild av systemet

För både lexikalisk analys och parsning av texten i ett dokument avsett för vårt programmeringsspråk använde vi oss av rdparse. Detta gjorde vi eftersom vi redan var bekanta vid det sedan TDP007 och för att vi ansåg att det hade varit för mycket arbete att konstruera en ny lexer/parser.

Innan lexern och parsern går igenom texten så sätts den ihop som en enda lång sträng för att den ska kunna läsas in som en kontinuerlig ström av text, istället för rad för rad (något vi hade problem med i början av detta projekt). Efter att denna sträng har bildats kallas codeParser.parse() med denna sträng som parametervärde. ”codeParser” är en instans av klassen Parser som definierar rdparse.

Därefter går lexern igenom all inkommande text för att bilda tokens utav de, vilket den gör genom att jämföra text som den läst in med våra reserverade nyckelord. Följande är en lista på dessa reserverade nyckelord, vilka vi representerar med reguljära uttryck:

- int
- float
- bool
- string
- array
- def
- end
- \d+ (för heltal)
- \d+.\d+ (för decimaltal)
- ”[^”]*” (för strängar)
- !=
- ==
- <=
- =>
- \+\+ (för inkrementering)
- \-\- (för dekrementering)
- &&
- \|\| (för logiskt eller)
- else if
- else
- for
- while
- return
- [a-zA-Z]+
- .
- ;

När lexern skapat tokens utav texten som uppkom i strömmen så försöker parsern matcha de med vår uppsättning av grammatiska regler. Dessa definieras enligt BNF-notation:

```
<statements> ::= <statements> <statement> | <statement>

<statement> ::= <definition> | <ariexpression> ";" | <variable> ";" |
                <logicexpression> ";" | <loop> | <controlstatement> |
                "return" <ariexpression> ";"

<definition> ::= <variable> "=" <ariexpression> ";" | <function_start> "("
                 <parameters> ")" <statements> "end"
```

```
<loop> ::= <looptype> "(" <definition> <logicexpression> ";" <ariexpression>
           ")" "{" <statements> "}" | <looptype> "(" <conditions> ")" "{"
           <statements> "}"

<looptype> ::= "for" | "while"

<controlstatement> ::= <controloperator> "{" <statements> "}" |
                       <controloperator> "(" <conditions> ")"
                       "{" <statements> "}" <controlstatement> |
                       <controloperator> "(" <conditions ")"
                       "{" <statements> "}"

<controloperator> ::= "if" | "else if" | "else"

<conditions> ::= <conditions> "&&" <logicexpression> | <conditions> "||"
                 <logicexpression> | <logicexpression>

<functionstart> ::= "def" <id>

<parameters> ::= <parameter> "," <parameters> | <parameter>

<parameter> ::= <variable>

<functioncall> ::= "split" "(" <string> ")" | "print" "(" <ariexpression>
                   ")" | <id> "(" <values> ")"

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

<value> ::= <ariexpression>

<logicexpression> ::= <ariexpression> <oplogic> <ariexpression>

<oplogic> ::= "==" | "!=" | "<=" | "=>" | "<" | ">"

<ariexpression> ::= <ariterm> "+" <ariexpression> | <ariterm> "-"
                    <ariexpression> | <id> "++" | <id> "--" |
                    <ariterm>

<ariterm> ::= <atom> "*" <ariterm> | <atom> "/" <ariterm> | <atom> "^"
              <ariterm> | <atom>

<atom> ::= <array> | <string> | <number> | <bool> | <functioncall> |
           <variable> | "(" <ariexpression> ")"

<number> ::= Integer | Float

<string> ::= /"[^"]*"/
```

```
<array> ::= <id> "[" <number> "]" | "[" <values> "]"

<bool> ::= /true/ | /false/

<variable> ::= <type> <id> | <id>

<type> ::= /int/ | /float/ | /bool/ | /string/ | /array/

<id> ::= /[a-zA-Z]+/
```

För varje matchning med ett <statement> eller <statements> så läggs den nod som skapats till i listvariabeln $lines. När syntaxträdet byggts upp så går codeParser:n igenom varje nod och evaluerar dem i den ordning som de skapats, vilket blir en rad-för-rad exekvering av koden i textdokumentet.

4.1 Klasser för noder i språket

Vid generering av syntaxträdet skapas flertal noder som representerar olika konstruktioner i vårt språk. Dessa noder representeras i sin tur av diverse klasser, så som FunctionNode, IntegerNode, etc. Varje nod har en initialize() funktion, ansvarig för att ställa upp noden med rätt värden på dess datamedlemmar, och en evaluate() funktion som är ansvarig för att utföra nodens uppgift.

Värdena som de instansieras med kommer från BNF-matchningarna. Olika noder har olika komplicerade evaluate() funktioner. IntegerNode, som är ansvarig för att representera heltal, har en evaluate() funktion som bara returnerar det som dess value-datamedlem är satt till. FunctionCall:s evaluate() funktion däremot gör ett flertal saker, som att till exempel hitta funktionen den kallar på i det globala scope:et, ställa upp dess parametrar med rätt värden, köra evaluate() för varje nod i funktionskroppen, osv.

Eftersom en nod kan ha en annan nod i sina datamedlemmar är det viktigt att noder har en gemensam bas, det vill säga att alla noder innehåller minst en evaluate() funktion utöver initialize().

De klasser som finns är:

- VariableNode
- ArithmeticNode
- IntegerNode
- FloatNode
- BoolNode
- BoolValueNode
- StringNode
- PrintNode
- SplitNode
- ArrayNode
- ArrayAccessorNode
- ArrayElement
- AssignmentNode
- FunctionNode

- FunctionCallNode
- IfStatementNode
- ElseStatementNode
- LogicNode
- AndNode
- OrNode
- ReturnNode
- ForLoopNode
- WhileLoopNode

4.2 Scope

Scope:s i vårt språk representeras av klassen Scope som har tre datamedlemmar: variables, functions och parent. Variables och functions är listor som innehåller variabler respektive funktioner som scope:et innefattar. Parent är det scope som ligger ”ovanför” det scope:et som instansieras. I codeparser.rb finns det två globala variabler som håller reda på det globala scope:et och det nuvarande scope:et: $global_scope och $current_scope.

Vårt språk är statiskt scope:at, och allt scoperelaterat sker vid parsning (funktionsanrop är ett undantag, uppletning av funktioner sker i runtime). När en matchning för ett <id> sker så letar CodeParser:n efter variabeln först i det nuvarande scope:et, och fortsätter med att leta i det ovanstående scope:et tills den når det globala scope:et.

Vi ansåg att scopehantering borde ske vid parsning då det parsern ska göra är att placera ut VariableNode:s på rätt plats i rätt ordning i syntaxträdet, vilket vi kände var lämpligt att göra vid parsning. Funktionsanrop fick vi däremot göra så att uppletning av en funktion skedde vid runtime, så att man kan anropa funktioner som inte än definierats.

5 Reflektion

Nu när vi har i princip nått slutet av kursen så kan vi se att resultatet stämmer överens för det mesta med den vision vi hade i början av kursen. Alla konstruktioner vi ville ha med finns med, däremot så fick for-loopar en annorlunda syntax i slutet än det vi bestämde i vår senaste språkspecifikation.

I ett av kodexemplena inkluderade vi dock f-strings, vilket är något som inte finns med i vårt språk, helt enkelt för att vi glömde bort att implementera dem. En annan sak som vi var osäkra på om vi skulle implementera funktionalitet för var hashar och listor, vi bestämde oss för att vi skulle implementera detta beroende på hur vi låg till tidsmässigt och i slutändan så implementerade vi funktionalitet för listor men inte hashar.

Användarvänligheten är något som hade behövts förbättras för att ta språket till en högre nivå. I nuläget har vi i princip inga fel meddelanden till användaren, detta gör att det blir svårare att förstå hur språket fungerar och det blir svårare för användaren att debugga koden.

Under projektets gång redigerades BNF-grammatiken beroende på vilka implementationer vi hade gjort. Detta medförde att vi vid olika tillfällen fick problem med BNF-matchningar. Problemen som kunde uppstå var att vi som exempel hade matchning av Int i regeln för ariexpression, detta gjorde att vi gick upp och ner i grammatik-trädet vilket skapade en extra IntegerNode vid matchningar. I efterhand kan det varit bättre att fokusera lite mer tid på strukturen av vår BNF, men eftersom detta var ett nytt koncept för oss så hade det blivit svårt att strukturera upp ett fungerade BNF-träd.

Vid variabler valde vi att spara dem i hashar med deras objekt-id som nyckel istället för variabelns namn och detta berodde på att vi vid implementation av variabler inte hade börjat arbeta med scopes, vilket gjorde att vi inte kunde skapa variabler med samma namn. När vi nu är klara och har skapat scopes så är detta sättet att använda objekt id onödigt då vi nu inte har en hash för alla variabler, utan en hash för varje specifikt scope.

Generellt sätt så är vi nöjda med allt vi gjorde. Att bygga upp BNF-grammatiken (regler och matchningar) var den svåraste delen i projektet och tog som tidigare förklarat många verisioner och mycket tid för att få klart. Däremot var klasserna för noder i språket och självaste koden i matchningar en del av projektet som gick relativt snabbt och var relaltivt enkelt. Anledningen till detta var förmodligen att vi i tidigare projekt och kurser gjort liknande programmering, exempel på detta är i TDP005 (spel projektet) där det går att jämföra implementationen av våra node klasser med klasserna för objekt/varelser i spelen.