Dunder

Dokumentation

Björn Gylling och Linus Karlsson

Innehållsförteckning

1 Inledning

Vi är två personer som går första året på Innovativ Programmering på Linköpings tekniska högskola. Vi fick som uppgift i kursen TDP019 att konstruera ett programmeringsspråk. Resultatet blev Dunder.

Dunder är en blandning mellan programmerings språken Ruby och C++ som vi har använt oss av i tidigare kurser.

Dunder är till för den som redan har vissa kunskaper inom programmering eller HTML och vill utöka sina kunskaper. Till Dunder finns en interaktiv tolk som kan köra kod i realtid, det gör att det är mycket lätt att testa olika kod snabbt och se resultatet.

2 Användarhandledning för Dunder

Dunder är en blandning mellan "traditionella" språk som till exempel C++ och enklare skriptspråk som Python och Ruby. Språket är anpassat till de som inte programmerat så mycket tidigare. Det är förhållandevis enkelt tack vare dynamiska variabler och en interaktiv tolk där programmeraren snabbt kan prova olika konstruktioner.

2.1 Komma igång

För att använda Dunder så behöver du först och främst installera Ruby. Därefter behöver du ladda ner Dunder. Projektet finns tillgängligt på github.com/bjorngylling/TDP019, klicka på *Download Source* uppe till höger för att ladda ner allt du behöver. Därefter kan du köra igenom de tester som finns med för att försäkra dig om att allt fungerar. Det görs enklast genom att köra filen `run_tests.rb` som ligger direkt i projektets huvudmapp. Den borde rapportera "`50 tests, 111 assertions, 0 failures, 0 errors`". Antalet test och assertions kan ha ändrats ifall fler har lagts till sedan detta dokumentet skrevs men så länge det är 0 failures och 0 errors så ska allt fungera. Därefter kan du köra Dunder genom att köra filen `dunder.rb` som ligger i mappen `lib/`. Till exempel så kommer detta kommandot skriva ut versionen "`ruby lib/dunder.rb -v`". För att köra en fil med Dunder-kod så kör du bara `dunder.rb` med filens sökväg som argument, alltså "`ruby lib/dunder.rb test/fixtures/factorial.dun`" till exempel.

2.2 Dunders interaktiva tolk

För att starta den interaktiva tolken skriv in detta kommandot "`ruby lib/dunder.rb -ip`". Den interaktiva tolken kan användas för att lätt testa kod och för att testa hur Dunder fungerar. I den interaktiva tolken kommer din kod att köras direkt, så det är ett väldigt snabbt sätt att skriva små tester. För de exempel som kommer lite senare så fungerar det perfekt att använda den interaktiva tolken.

3 Skriva kod i Dunder

Ett enkelt exempel på ett litet program är att skriva ut "Hello world", i Dunder kan man göra det på ett lätt sätt:

```
print "Hello world"
```

Man kan även lägga till kommentarer för att förtydliga sin kod. Det finns tre olika sätt att skriva kommentarer:

```
print "Hello world" # End-line kommentar
print "Goodbye world" // Detta är också en kommentar
\* Denna kommentar
kan vara på flera rader
*\
```

3.1 Variabler och tilldelning

Senare kan man tilldela variabler både siffror och text på detta sättet:

```
x = 10 # Tilldelar x värdet 10
y = 5 # Tilldelar y värdet 5
z = "Hello world" # Tilldelar z en sträng
```

Man kan sedan använda dessa variabler för att göra beräkningar och skriva ut variabler:

```
c = x - y # c = 10 – 5
print c # Skriver ut c
```

3.2 If-satser

Dunders if-sats fungerar så att den kommer köra koden inom kod-blocket om det som står i uttrycket är sant, om det inte är det och det finns en else-sats så kommer den koden som finns där att köras.

I Dunder är det inga problem att ha flera if-satser i varandra.

En enkel if-sats kan se ut så här:

```
if( x == 10) { # ifall x är lika med 10 så kommer "Hello world" skrivas
ut
  print "Hello world"
}
```

Det uttryck som står innanför paranteserna kommer att evalueras och om det är sant så kommer de satser som står inom {} att köras.
Operatorer som kan användas är:

- Lika med `x == y`
- Inte lika med `x != y`
- Större än `x > y`
- Större lika med `x >= y`
- Mindre än `x < y`
- Mindre lika med `x <= y`

Man kan även göra en if-else sats och den kan se ut så här:

```
x = 4
if(x==5) {
  print "Hello world"
}
else {
  print "World hello"
}
```

`Else`-satsen kommer bara köras om uttrycket i `if` är falskt. Detta gör att man kan göra olika saker beroende på vad som händer.

3.3 While-satser

En while-sats i Dunder är lik en i många andra språk. Den kommer att loopas så länge det som står i uttrycket är sant.

En enkel while-sats kan se ut så här:

```
x = 0
while( x != 5) { # Denna loop kommer köras tills x == 5
  x = x + 1
  print "Hello world"
}
```

Så klart så fungerar alla operatorer som funkar på if-satser i while-satser.

3.4 Funktioner

En funktion i Dunder fungerar som i de flesta andra språk.

Om det inte finns någon return i funktionen så kommer sista raden i funktionen returneras. Man kan använda return för att vara mer tydlig med vad som returneras och return kan även användas inne i en if-sats.

En enkel funktion kan se ut så här:

```
def say_hello() {
  print "Hello world"
}
say_hello # Kallar på funktionen, funktionen kommer då köras
```

Man kan även använda parametrar till funktioner:

```
def say_something(word) {
 print word
}
say_something("Hello!") # Kallar på funktionen med argumentet Hello!
```

En funktion kan ta in flera parametrar:

```
def addition(x,y) {
 return x + y
}
c = addition(3, 4) # Tilldelar c resultatet av addition funktionen.
```

3.5 Lambda-funktioner

Det går även att deklarera så kallade lambda-funktioner i Dunder. Detta visas lättast med ett exempel:

```
foo = { |var| var + 10 }
```

Detta ger samma resultat som följande:

```
def foo(var)
  var + 10
end
```

Det går även att använda lambda-funktioner som argument till andra funktioner, se följande exempel:

```
foo = { |var| var + 10 }

def bar(function) {
  function(10) + 2
}

bar(foo)
```

Denna kod kommer returnera 22.

3.6 Två små program i Dunder

Nu använder vi oss av de tekniker som visats upp för att skriva två stycken enkla program i Dunder.

3.6.1 Fakultet

```
5! = 1*2*3*4*5 = 120
```

Här är ett litet program som kan räkna ut fakulteten på ett tal. För att göra det här programmet används både if- och while-satser. I början så är det mycket viktigt att kontrollera så att det talet som skickas inte är lika med eller mindre än 0 för det fungerar inte att ta fakulteten på ett sådant tal. Om det är för litet så skriver vi helt enkelt ut att talet är för litet och sedan returnerar vi 0 så att funktionen inte fortsätter till nästa del. Vi sätter sedan x till 1 och answer till 1 för att om x skulle vara 0 eller om answer skulle vara 0 skulle inte det funka eftersom då skulle man hela tiden multiplicera med 0. Sedan kommer funktionen gå vidare till while-satsen, där skriver vi att den ska loopa medan x är mindre eller lika med numret som skickas in. Sedan kommer x att multiplicera sig med awnser och sedan kommer vi öka x med 1. Detta kommer göras ända tills vi har det rätta svaret. När vi sedan har det rätta svaret så returnerar vi answer.

Testa nu att skriva in koden själv, svaret på 5! är 120.

Här är koden:

```
def factorial(number) {
  if(number <=0) { # Kollar om number är giltigt
    print "Talet är för litet"
    return 0
  }
  x = 1
  answer = 1
  while(x <= number) {
    answer = answer * x # Multiplicerar det numret med gamla faktorn
    x = x + 1
  }
  return answer # Returnerar svaret
}
```

3.6.2 Fibonacci

Några fibonacci-tal: `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144`

Nu kommer du få se koden i Dunder för att skapa en funktion som räknar ut det fibonacci-tal som anropas med funktionen. I det här exemplet ska vi använda funktioner rekursivt, det vill säga att funktionen kallar på sig själv för att räkna ut det man vill. Detta passar sig väldigt bra när det gäller fibonacci-tal. Vi börjar med att kontrollera ifall talet som skickas in är ett tal som går att räkna ut. Det måste alltså vara större än 1, om det inte är det så skickar vi tillbaka det talet som skickades in. Detta hjälper så att inte rekursiviteten går för långt tillbaka. Vi vill inte att den ska gå under tal 1. Efter det kommer funktionen använda sig av rekursivitet för att få fram det tal som man letat efter. Testa nu själv!

```
def fib(n) {
  if (n <= 1) { # Fib(1) eller lägre skall returnera sitt eget värde
    return n
  } else {
    return fib(n-1)+fib(n-2) # Här används rekursiviteten.
  }
}
```

4 Systemdokumentation för Dunder

Dunder är en blandning mellan C++, Ruby och Python. Till största del är Dunder likt Ruby men med vissa inslag av C++s strikta syntax med paranteser och klammrar. I Dunder är variablerna dynamiska som i Ruby och Python, det gör att programmeraren inte behöver hålla reda på vilken typ som variabeln som han arbetar med. Dunder har även en interaktiv tolk som gör det enkelt att testa korta kod i språket. Dunder är inte ett kompilerat språk utan använder sig av en interpretator. Dunder är lätt att lära sig om man har erfarenheter av tidigare programmering men kan även vara ett bra språk att börja med som helt ny.

4.1 Teknik och översiktligt beskrivning av hur Dunder fungerar

Vi har använt oss av rdparser för att skapa ett parse-träd av noder. Varje nod är en instans av en klass. Det som händer när man skickar in kod till Dunder::Parser.new.parse är att först så plockas eventuella kommentarer bort från koden. Sedan plockas nyradstecken bort på de platser de inte behövs för parsning av språket. Sedan skickas koden till rdparsern där den delas upp i tokens för att sedan gås igenom och bli vårat parse-träd. Man får alltså tillbaka ett träd som man sedan kan köra #eval på för att själva trädet ska exekveras och returna resultatet av koden. Det är alltså inte förräns #eval körs på parse-trädet som själva koden exekveras. I vanliga fall så görs det direkt efter parsning.

4.2 Förklaring av variabelscope i Dunder

Variabelhantering i Dunder är ganska "öppet". Dunder har ett globalt scope i grunden, där ligger alla variabler som inte finns i någon funktion. När sedan en funktion skapas så skapas även ett eget scope till den funktionen och det nya scope kommer då länkas ihop med scope som ligger "över". När sedan funktionen körs så kommer funktionen först leta igenom sitt eget scope för att hitta variabeln som den söker och finns den inte där så börjar den leta i scopet som ligger över. Skulle inte variabeln finnas så skapas en ny i funktionens scope. Detta gör att om det globala scopet har en variabel vid namnet x och funktionen change har en variabel som också heter x kommer det alltså i funktionens scope finnas en variabel som heter x. Det gör att sen när funktionen change ändrar på x kommer den enbart ändra för funktionens x. En funktion kan alltså inte ändra på en variabel som ligger i högre scope utan bara läsa ifrån dem. Här visas ett exempel på hur en funktion hanterar scope:

```
x = 10
def change() {
  x = 5
}
change()
```

Efter denna kod har exekverats kommer så kommer x fortfarande vara 10. Här kommer ett exempel på hur scope funkar i en while-loop:

```
x = 0
while(x<5) {
  x = x + 1
}
```

Efter denna kod har exekverats så kommer x vara 5.

4.2.1 Statisk bindning

En annan intressant egenskap i Dunder är att funktioner har så kallad statiskt bindning. Det betyder att det scope där funktionen finns deklarerade kommer alltid vara det scope över funktionen. Detta visas lättast med ett exempel:

```
var = 10

def foo(x) {
  var = x
  bar()
}

def bar() {
  print var
}

foo(5)
```

Utskriften av variabeln var i funktionen bar kommer alltså inte leta i scopet för funktionen foo utan kommer gå direkt till det globala scopet där bar är deklarerad. Detta innebär till exempel en stor fördel i hastighet vid rekursiva funktioner. Det funktionsanrop som körs längst ner i den rekursiva kedjan kommer inte behöva gå igenom alla steg i den rekursiva kedjan för att hitta deklarationen av funktionen själv.

4.3 Filstruktur

```
doc/
  grammar_lexical.txt
  grammar_syntax.txt
lib/
  dunder/
    helpers.rb
    nodes.rb
    object.rb
    parse.rb
  rdparse/
    rdparse.rb
  dunder.rb
test/
  fixtures/
    *.dun
  nodes_test.rb
  parser_test.rb
  test_helpers.rb
VERSION
README.md
run_tests.rb
```

Här följer en kort beskrivning av innehållet i varje fil.

- `doc/` här finns språkets grammatik.
- `lib/` här hittar du själva koden.
    - `lib/dunder.rb` är själva "ingångspunkten". Det är denna filen som användaren arbetar mot. Den tar in antingen en fil som förväntas innehålla kod skriven i Dunder, koden körs då. Den kan även ta in en av två flaggor. Flaggan `-v` skriver endast ut vilken version det är, alltså det som står i filen `VERSION`. Den andra flaggan som är lite intressantare är `-ip` som står för *interactive parser*. Kör man Dunder med den flaggan så startas en interaktiv tolk där man kan skriva Dunder-kod som exekveras i realtid. Detta är ett ganska intressant verktyg om programmeraren vill prova en viss funktionalitet snabbt och dynamiskt.
    - `lib/rdparse/rdparse.rb` är den rdparser som vi fått tillgång till i TDP015, den är helt orörd för att undvika att krångel.
    - `lib/dunder/helpers.rb` - här finns två hjälpfunktioner som hanterar läsning och skrivning i våra scopes.
    - `lib/dunder/nodes.rb` - här finns alla våra noder. Mer om dem längre fram.
    - `lib/dunder/object.rb` - den här filen bygger endast ut Ruby-klassen Object med en funktion för att kontrollera om något är en nod eller inte.
    - `lib/dunder/parse.rb` - här finns alla våra parse-regler, kod för att bearbeta koden innan den skickas till parsning, till exempel plocka bort kommentarer. Här finns också koden för Dunders interaktiva tolk.
- `test/` innehåller alla våra testfall.
    - `nodes_test.rb` för våra noder.
    - `parser_test.rb` för där vi testar parsning och körning av faktisk kod. `parser_test.rb` är egentligen den mest intressant av de två.
    - `test/test_helpers.rb` innehåller en del hjälpfunktioner mest för `nodes_test.rb` för att inte behöva upprepa sig så mycket i själva testerna.
    - `test/fixtures/` - innehåller filer med Dunder-kod som används av testerna.
- `VERSION` – versionsnummer
- `README.md` - enkel beskrivning av språket på engelska.
- `run_tests.rb` - kör alla våra tester. ("`ruby run_tests.rb`")

4.4 Dunder::Nodes

Dunder består av en mängd noder, allt ifrån en enkel integer till en funktionsdefinition. Jag tänker inte gå igenom alla noder men tänkte ta upp och beskriva några utav de mer intressanta.

Först och främst så har vi StatementList som innehåller en lista av Statement, när en StatementList evalueras så går den igenom sin lista och evaluerar varje statement i tur och ordning.

Sen har vi repetering, WhileStatement. Den tar in två variabler, ett villkor och en StatementList. När WhileStatement sedan evalueras så körs StatementList om och om igen sålänge villkoret evalueras till ett sanningsvärde. Om inte något utav uttrycken i StatementList påverkar villkoret så att det blir falskt så kommer alltså loopen att köras.

En annan intressant nod är FunctionCall. Det är den mest avancerade noden i Dunder. När ett funktionsanrop evalueras så letar den först upp definitionen av funktionen som skall anropas, sedan plockar den ut parametrarna från funktionsdefinitionen och binder ihop dem med de argument som skickats till funktionsanropet och skapar ett nytt variabelscope av det. Detta scope länkar till scopet utanför funktionen fast inte med det vanliga "`PARENTSCOPE`" nyckelordet utan ett specifikt nyckelord för funktioner. Detta är för att vi inte ska kunna ändra på värden i variabler som ligger i högre scope utan endast läsa dem. Till sist körs den StatementList som finns definerad i funktionsdefinitionen. Hittas ett ReturnStatement så returneras det och annars så returneras det sista Statementet i funktionen.

5 Erfarenheter och reflektion

När vi byggde grammatiken för språket så lärde vi oss att tyda andra språks gramatiker. Detta kommer vi ha användning för senare om vi någon gång kommer behöva se närmare på andra språk. Vi har även förstått mer hur komplicerat det är att bygga ett språk.

Vi ändrade ganska mycket i grammatiken för uttryck då vi från början inte tänkt på att vi behövde prioritet, till exempel att multiplikation har högre prioritet än addition. Det kunde vi löst på två olika sätt, antingen använda järnvägsalgoritmen eller bygga om Dunders grammatik så den tog hand om det. Vi valde att göra ändringarna i grammatiken för det kändes mest logiskt fall. När vi gjorde det så hade vi väldigt stor nytta av Pythons grammatik som hanterar prioritet på precis samma sätt. Språkets version blir dock lite nerbantad då vi inte har stöd för till exempel "upphöjt till".

5.1 Hantering av variabler

Vi har fått lära oss mycket om hur så kallade "scope" eller variabeltabeller fungerar på olika sätt beroende på vad man vill ha dem till. I Dunder har vi det till för att alla variabler ska läsas på rätt sätt. Vi skapar ett nytt scope för varje enskild funktion så att den har en lista på varje variabel som skapas i den funktionen, skulle inte variabeln finnas i funktionen så kommer den söka i det scopet som ligger ett steg högre upp i ordningen, till exempel det globala scopet som inte tillhör någon funktion utan som har en lista på alla variabler utanför alla funktioner. När man skapar scopet för en funktion så gäller det också att binda parametrar med de argument som skickats till funktionen och så att säga förbereda funktionens scope med dessa så de finns tillgängliga när funktionens kod körs.

5.2 Ruby

Vi har lärt oss att använda regular expressions på ett mycket mer bättre och avancerat sätt för att plocka ut de data som vi vill ha med från olika saker, till exempel att ta bort de kommentarer som programmeraren skriver i programmet eller som att plocka ut viktiga ord som def, if, while osv.

5.3 Testning

Vi har lärt oss att testfall är väldigt användbara när man sitter och gör program eftersom vi kan testa alla saker vi har med direkt och snabbt. Det gör att vi slipper att skriva om alla tester som man kan komma på och det gör också att vi inte missar något som vi brukar testa. Vi valde att använda oss av Rubys inbyggda test-ramverk test-unit. Så här i efterhand hade det varit intressant att titta på något lite mer avancerat ramverk och lära sig det, till exempel Rspec.

5.4 Allmänt

Vi har lärt oss att göra egna kod-exempel, det är inte det lättaste för man ska helst skapa sånna exempel som är lätta att förstå med även lär ut till användaren.

6 Referenser

Vi har inte använda några direkta referenser för skrivandet av detta dokument. Vi har haft en del resurser till hjälp när vi programmerade Dunder. Utöver diskussion med Anders Haraldsson så har vi använt följade resurser.

Bra hemsida för att testa reguljära uttryck. http://rubular.com

Rubys officiella dokumentation. http://ruby-doc.org

Pythons grammatik. http://docs.python.org/release/2.5.2/ref/grammar.txt

Den parser vi använt oss av ifrån en tidigare kurs (TDP007).
http://www.ida.liu.se/~TDP007/material/examples/rdparse.rb

Grammatik

```
digit ::= "0"..."9"

letter ::= lowercase | uppercase

lowercase ::= "a"..."z"

uppercase ::= "A"..."Z"

identifier ::= lowercase [ letter | digit | "_" ]* |
               "_" [ letter | digit | "_" ]+

end_line_comment ::= [ "//" | "#" ] anything "\n"

multi_line_comment ::= "/*" anything "*/"

string ::= '"' anything except " '"' | "'" anything except ' "'"

statement_list ::= [ [statement ] statement_terminator ]+

statement_terminator ::= "\n" | ";"

statement ::= compound_statement |
              expression |
              return_statement

return_statement ::= "return" expression

compound_statement ::= if_statement |
                       while_statement |
                       function_def

block_start ::= "{"

block_end ::= "}"

if_statement ::= "if" "(" expression ")" block_start statement_list
block_end |
                 "if" "(" expression ")" block_start statement_list
block_end
                  "else" block_start statement_list block_end

while_statement ::= "while" "(" expression ")" block_start statement_list
block_end

function_def ::= "def" identifier parameters block_start statement_list
block_end

lambda_function ::= block_start "|" parameter_list? "|" statement_list
block_end

parameters ::= "(" parameter_list ")" |
               "(" ")"

parameter_list ::= [identifier ","]* identifier

function_call ::= identifier arguments
```

```
arguments ::= "(" expression_list ")" |
              "(" ")"

expression_list ::= [expression ","]* expression

expression ::= assignment_expression |
               comparison |
               lambda_function

comparison ::= [a_expr comp_operator]* a_expr

a_expr ::= [m_expr ["+" | "-"]]* m_expr

m_expr ::= [u_expr ["*" | "/"]]* u_expr

u_expr ::= ["+" | "-"]? primary

primary ::= function_call |
            boolean |
            number |
            string |
            indentifier

comp_operator ::= "==" | "<=" | ">=" | ">" | "<" | "!="

assignment_expression ::= identifier "=" expression

number ::= integer | float

integer ::= digits

digits ::= digit+

float ::= [ digits ] "." digits
```

```ruby
#!/usr/bin/ruby
# coding: utf-8

####
# dunder.rb
# http://github.com/bjorngylling/TDP019
# Part of the Dunder language project by Björn Gylling and Linus Karlsson
# This file is the "entrypoint" for the end-user. It can either start the
# interactive parser or parse code from a file. Runs from a commandline.

# Require all files in lib/dunder
$:.unshift File.dirname(__FILE__)
Dir["#{File.dirname(__FILE__)}/dunder/*.rb"].each do
  |format| require "dunder/#{File.basename format}"
end

def evaluate_file(file_name)
  global_scope = Hash.new
  file = read_file_to_string(file_name)
  puts Dunder::Parser.new.parse(file).eval(global_scope)
end

def read_file_to_string(file_name)
  File.open(file_name, "r") { |f| f.read.gsub(/\r\n/, "\n") }
end

options = {:v => Proc.new { puts read_file_to_string("VERSION") },
           :ip => Proc.new { Dunder::Parser.new.interactive_parser }}

if !ARGV.empty?
  flag = ARGV.first.delete("-").to_sym
  if options.has_key? flag
    options[flag].call
  else
    evaluate_file ARGV.first
  end
elsif($0.include? "dunder.rb")
  puts "Dunder help:
  Flags:
    -v  Prints the version
    -ip Starts the interactive parser

  Pass a file with Dunder-code as the argument and Dunder will run the code in that file."
end
```

```ruby
#!/usr/bin/env ruby
# coding: utf-8

# 2010-02-11 New version of this file for the 2010 instance of TDP007
#   which handles false return values during parsing, and has an easy way
#   of turning on and off debug messages.

require 'logger'

module Rdparse

  class Rule

    # A rule is created through the rule method of the Parser class, like this:
    #   rule :term do
    #     match(:term, '*', :dice) {|a, _, b| a * b }
    #     match(:term, '/', :dice) {|a, _, b| a / b }
    #     match(:dice)
    #   end

    Match = Struct.new :pattern, :block

    def initialize(name, parser)
      @logger = parser.logger
      # The name of the expressions this rule matches
      @name = name
      # We need the parser to recursively parse sub-expressions occurring
      # within the pattern of the match objects associated with this rule
      @parser = parser
      @matches = []
      # Left-recursive matches
      @lrmatches = []
    end

    # Add a matching expression to this rule, as in this example:
    #   match(:term, '*', :dice) {|a, _, b| a * b }
    # The arguments to 'match' describe the constituents of this expression.
    def match(*pattern, &block)
      match = Match.new(pattern, block)
      # If the pattern is left-recursive, then add it to the left-recursive set
      if pattern[0] == @name
        pattern.shift
        @lrmatches << match
      else
        @matches << match
      end
    end

    def parse
      # Try non-left-recursive matches first, to avoid infinite recursion
      match_result = try_matches(@matches)
      return nil if match_result.nil?
      loop do
        result = try_matches(@lrmatches, match_result)
        return match_result if result.nil?
        match_result = result
      end
```

```
    end

    private

    # Try out all matching patterns of this rule
    def try_matches(matches, pre_result = nil)
      match_result = nil
      # Begin at the current position in the input string of the parser
      start = @parser.pos
      matches.each do |match|
        # pre_result is a previously available result from evaluating expressions
        result = pre_result ? [pre_result] : []

        # We iterate through the parts of the pattern, which may be e.g.
        #   [:expr,'*',:term]
        match.pattern.each_with_index do |token,index|

          # If this "token" is a compound term, add the result of
          # parsing it to the "result" array
          if @parser.rules[token]
            result << @parser.rules[token].parse
            if result.last.nil?
              result = nil
              break
            end
            @logger.debug("Matched '#{@name} = #{match.pattern[index..-1].inspect}'")
          else
            # Otherwise, we consume the token as part of applying this rule
            nt = @parser.expect(token)
            if nt
              result << nt
              if @lrmatches.include?(match.pattern) then
                pattern = [@name]+match.pattern
              else
                pattern = match.pattern
              end
              @logger.debug("Matched token '#{nt}' as part of rule '#{@name} <= #{pattern.
inspect}'")
            else
              result = nil
              break
            end
          end
        end
        if result
          if match.block
            match_result = match.block.call(*result)
          else
            match_result = result[0]
          end
          @logger.debug("'#{@parser.string[start..@parser.pos-1]}' matched '#{@name}' and
generated '#{match_result.inspect}'") unless match_result.nil?
          break
        else
          # If this rule did not match the current token list, move
          # back to the scan position of the last match
          @parser.pos = start
```

```
      end
    end

    return match_result
  end
end

class Parser

  attr_accessor :pos
  attr_reader :rules, :string, :logger

  class ParseError < RuntimeError
  end

  def initialize(language_name, &block)
    @logger = Logger.new(STDOUT)
    @lex_tokens = []
    @rules = {}
    @start = nil
    @language_name = language_name
    instance_eval(&block)
  end

  # Tokenize the string into small pieces
  def tokenize(string)
    @tokens = []
    @string = string.clone
    until string.empty?
      # Unless any of the valid tokens of our language are the prefix of
      # 'string', we fail with an exception
      raise ParseError, "unable to lex '#{string}'" unless @lex_tokens.any? do |tok|
        match = tok.pattern.match(string)
        # The regular expression of a token has matched the beginning of 'string'
        if match
          @logger.debug("Token #{match[0]} consumed")
          # Also, evaluate this expression by using the block
          # associated with the token
          @tokens << tok.block.call(match.to_s) if tok.block
          # consume the match and proceed with the rest of the string
          string = match.post_match
          true
        else
          # this token pattern did not match, try the next
          false
        end # if
      end # raise
    end # until
  end

  def parse(string)
    # First, split the string according to the "token" instructions given.
    # Afterwards @tokens contains all tokens that are to be parsed.
    tokenize(string)

    # These variables are used to match if the total number of tokens
    # are consumed by the parser
```

```
      @pos = 0
      @max_pos = 0
      @expected = []
      # Parse (and evaluate) the tokens received
      result = @start.parse
      # If there are unparsed extra tokens, signal error
      if @pos != @tokens.size
        raise ParseError, "Parse error. expected: '#{@expected.join(', ')}', found '#{@tokens
[@max_pos]}'\n"
      end
      return result
    end

    def next_token
      @pos += 1
      return @tokens[@pos - 1]
    end

    # Return the next token in the queue
    def expect(tok)
      t = next_token
      if @pos - 1 > @max_pos
        @max_pos = @pos - 1
        @expected = []
      end
      return t if tok === t
      @expected << tok if @max_pos == @pos - 1 && !@expected.include?(tok)
      return nil
    end

    def to_s
      "Parser for #{@language_name}"
    end

    private

    LexToken = Struct.new(:pattern, :block)

    def token(pattern, &block)
      @lex_tokens << LexToken.new(Regexp.new('\\A' + pattern.source), block)
    end

    def start(name, &block)
      rule(name, &block)
      @start = @rules[name]
    end

    def rule(name,&block)
      @current_rule = Rule.new(name, self)
      @rules[name] = @current_rule
      instance_eval &block
      @current_rule = nil
    end

    def match(*pattern, &block)
      @current_rule.send(:match, *pattern, &block)
    end
```

```
    end

end
```

```ruby
# coding: utf-8

####
# parse.rb
# http://github.com/bjorngylling/TDP019
# Part of the Dunder language project by Björn Gylling and Linus Karlsson
# This file contains the parse-rules for the rdparser. It also contains
# the code that runs the interactive parser. This parser builds a parse-tree
# of nodes.

require "lib/rdparse/rdparse.rb"

module Dunder

  class Parser
    include Dunder::Nodes

    def initialize
      @dunder_parser = Rdparse::Parser.new("dunder") do
        extend Dunder::Nodes

        # Strings
        token(/'[^']*'/) { |t| DString.new t.delete("'") }
        token(/"[^"]*"/) { |t| DString.new t.delete('"') }

        # Remove whitespace, except newlines since
        # they are statement_terminators
        token(/[ \t]+/)

        # Statment terminators
        token(/[\r\n;]/) { |t| t }

        token(/\w+|==|<=|>=|!=|\+|-|\*|<|>|\/|=|\{|\}|\(|\)|\.|\||,/) { |t| t }

        start :statement_list do
          match(:statement, :statement_terminator, :statement_list) do
            |a, _, b|
            b = a + b
          end
          match(:statement, :statement_terminator)
          match(:statement)
        end

        rule :statement_terminator do
          match("\n")
          match(";")
        end

        rule :statement do
          match(:return_statement) { |a| StatementList.new a }
          match(:print_statement) { |a| StatementList.new a }
          match(:compound_statement) { |a| StatementList.new a }
          match(:expression) { |a| StatementList.new a }
        end

        rule :return_statement do
          match("return", :expression) do |_, expression|
```

```
      ReturnStatement.new expression
    end
  end

  rule :print_statement do
    match("print", :expression) do |_, string|
      PrintStatement.new string
    end
  end

  rule :compound_statement do
    match(:if_statement)
    match(:while_statement)
    match(:function_def)
  end

  rule :block_start do
    match('{')
  end

  rule :block_end do
    match('}')
  end

  rule :if_statement do
    match("if", "(", :expression, ")", :block_start,
            :statement_list,
          :block_end,
          "else", :block_start,
            :statement_list,
          :block_end) do
      |_, _, condition, _, _, stmt_list, _, _, _, else_stmt_list, _|
      IfStatement.new condition, stmt_list, else_stmt_list
    end
    match("if", "(", :expression, ")", :block_start,
            :statement_list,
          :block_end) do |_, _, condition, _, _, stmt_list, _|
      IfStatement.new condition, stmt_list
    end
  end

  rule :while_statement do
    match("while", "(", :expression, ")", :block_start,
            :statement_list, :block_end) do
      |_, _, condition, _, _, stmt_list, _|
      WhileStatement.new condition, stmt_list
    end
  end

  rule :function_def do
    match("def", :identifier, :parameters, :block_start,
            :statement_list, :block_end) do
      |_, name, parameters, _, stmt_list, _|
      FunctionDefinition.new name, parameters, stmt_list.list
    end
  end
```

```
    rule :lambda_function do
      match(:block_start, '|', :parameter_list, '|', :statement_list, :block_end) do
        |_, _, parameters, _, stmt_list, _|
        FunctionDefinition.new nil, parameters, stmt_list.list
      end
      match(:block_start, '|', '|', :statement_list, :block_end) do
        |_, _, _, stmt_list, _|
        FunctionDefinition.new nil, [], stmt_list.list
      end
    end

    rule :parameters do
      match('(', :parameter_list, ')') { |_, list, _| list}
      match('(', ')') { |_, _| [] }
    end

    rule :parameter_list do
      match(:identifier, ",", :parameter_list) do
        |identifier, _, params|
        params + [identifier]
      end
      match(:identifier) { |identifier| [identifier] }
    end

    rule :function_call do
      match(:identifier, :arguments) do |name, args|
        Dunder::Nodes::FunctionCall.new name, args
      end
    end

    rule :arguments do
      match('(', :expression_list, ')') { |_, list, _| list }
      match('(', ')') { |_, _| [] }
    end

    rule :expression_list do
      match(:expression, ',', :expression_list) do
        |expression, _, list|
        list + [expression]
      end
      match(:expression) { |expression| [expression] }
    end

    rule :expression do
      match(:assignment_expression)
      match(:comparison)
      match(:lambda_function)
    end

    rule :comparison do
      match(:a_expr, :comp_operator, :a_expr) do |lh, op, rh|
        Comparison.new lh, op, rh
      end
      match(:a_expr)
    end

    rule :a_expr do
```

```
    match(:a_expr, "+", :m_expr) do | lh, _, rh |
      Addition.new lh, rh
    end
    match(:a_expr, "-", :m_expr) do | lh, _, rh |
      Subtraction.new lh, rh
    end
    match(:m_expr)
  end

  rule :m_expr do
    match(:m_expr, "*", :u_expr) do | lh, _, rh |
      Multiplication.new lh, rh
    end
    match(:m_expr, "/", :u_expr) do | lh, _, rh |
      Division.new lh, rh
    end
    match(:u_expr) { |a| a }
  end

  rule :u_expr do
    match("+", :primary)
    match("-", :primary) { |_, a| a.negative }
    match(:primary)
  end

  rule :primary do
    match(:function_call)
    match(:boolean)
    match(:number)
    match(:string)
    match(:identifier) do |name|
      Variable.new name
    end
  end

  rule :comp_operator do
    match('==')
    match('<=')
    match('>=')
    match('>')
    match('<')
    match('!=')
  end

  rule :assignment_expression do
    match(:identifier, '=', :expression) do |identifier, _, value|
      VariableAssignment.new identifier, value
    end
  end

  rule :identifier do
    match(/[a-z][A-Za-z0-9_]*/)
    match(/_[A-Za-z0-9_]+/)
  end

  rule :boolean do
    match('true') { DBoolean.new true }
```

```
      match('false') { DBoolean.new false }
    end

    rule :number do
      match(:float)
      match(:integer)
    end

    rule :integer do
      match(:digits) { |a| DInteger.new a }
    end

    rule :float do
      match(:digits, ".", :digits) { |a, _, b | DFloat.new a << "." << b }
      match(".", :digits) { |_, b | DFloat.new "0" << "." << b }
    end

    rule :digits do
      match(:digits, :digit) { |a, b| a << b }
      match(:digit)
    end

    rule :digit do
      match(/[0-9]/)
    end

    rule :string do
      match(DString)
    end

  end

  log false
end

def parse(code)
  code = remove_comments_in(code)
  code = remove_unwanted_newlines_in(code)

  @dunder_parser.parse(code)
end

def interactive_parser
  global_scope = Hash.new
  code = ""
  nested_blocks = 0

  puts "Dunder interactive parser. Type exit to quit."
  while(true)
    print "Dunder> "
    line = STDIN.gets
    break if line.chomp == "exit"

    code += line

    # Check if the line includes {, then there's more coming
    # so don't parse the code yet.
```

```
        if line.include? "{"
          nested_blocks += 1
          next
        end

        if line.include? "}"
          nested_blocks -= 1
        end

        if nested_blocks == 0
          result = parse(code).eval(global_scope)
          print "=> "
          puts result
          code = ""
        end
      end
    end

    def remove_comments_in(code)
      code = code.gsub /#.*$/, ""
      code = code.gsub /\/\/.*$/, ""
      code = code.gsub /\/\*(\n|.)*\*\//, ""
    end

    def remove_unwanted_newlines_in(string)
      string.gsub! /;[ \t]*\n/, "\n"
      string.gsub! /^[ \t]*\n/, ""
      string.gsub! /\{[ \t]*\n/, "{"
      string.gsub! /\|\n/, "|"
      string.gsub /\}[ \t]*\n[ |\t]*else/, "} else"
    end

    def log(state = true)
      if state
        @dunder_parser.logger.level = Logger::DEBUG
      else
        @dunder_parser.logger.level = Logger::WARN
      end
    end
  end

end
```

```ruby
# coding: utf-8

####
# helpers.rb
# http://github.com/bjorngylling/TDP019
# Part of the Dunder language project by Björn Gylling and Linus Karlsson
# This file contains helper-methods for the nodes

module Dunder
  module Helpers

    def look_up(name, scope)
      if scope.include?(name)
          return scope[name]
        else
          if scope.has_key? "PARENTSCOPE"
            # Go up one scope and look for it there
            return look_up(name, scope["PARENTSCOPE"])
          elsif scope.has_key? "OUTSIDE_FUNCTION_DEF"
            # Go up one scope and look for it there
            return look_up(name, scope["OUTSIDE_FUNCTION_DEF"])
          else
            return false
          end
        end
    end

    def assign(scope, name, value)
      scope[name] = value unless scope_assignement(scope, name, value)
    end

    def scope_assignement(scope, name, value)
      if scope.has_key? name
        scope[name] = value
        return true
      elsif scope.has_key? "PARENTSCOPE"
        # Check the variable exists in parent-scope, if it does we update it
        return scope_assignement(scope["PARENTSCOPE"], name, value)
      else
        return false
      end
    end

    def build_frame(parameters, arguments)
      Hash[*parameters.zip(arguments).flatten]
    end

  end
end
```

```ruby
# coding: utf-8

####
# nodes.rb
# http://github.com/bjorngylling/TDP019
# Part of the Dunder language project by Björn Gylling and Linus Karlsson
# This file contains all the nodes used to build the parse-tree and
# evaluate it.

module Dunder
  module Nodes
    @return = nil

    class Node
      include Dunder::Helpers

    end

    class StatementList < Node
      def initialize(statement)
        if statement.kind_of? Array
          @statement_list = statement
        else
          @statement_list = [statement]
        end
      end

      def +(statement_list)
        self.class.new @statement_list + statement_list.list
      end

      def list
        @statement_list
      end

      def eval(scope)
        result = nil
        @statement_list.each do |statement|

          result = statement.eval(scope)

          if statement.kind_of? Dunder::Nodes::ReturnStatement
            result = statement
            break
          end
          if result.kind_of? Dunder::Nodes::ReturnStatement
            break
          end

        end

        return result
      end
    end

    class DString < Node
      def initialize(value)
```

```ruby
    @value = value
  end

  def eval(scope)
    @value
  end
end

class DInteger < Node
  def initialize(value)
    @value = value.to_i
  end

  def eval(scope)
    @value
  end

  def negative
    self.class.new -@value
  end
end

class DFloat < Node
  def initialize(value)
    @value = value.to_f
  end

  def eval(scope)
    @value
  end

  def negative
    self.class.new -@value
  end
end

class DBoolean < Node
  def initialize(value)
    if value == nil || value == 0 || value == false
      @value = false
    else
      @value = true
    end
  end

  def eval(scope = Hash.new)
    @value
  end
end

class Addition < Node
  def initialize(lh, rh)
    @lh, @rh = lh, rh
  end

  def eval(scope)
    @lh.eval(scope) + @rh.eval(scope)
```

```ruby
  end
end

class Subtraction < Node
  def initialize(lh, rh)
    @lh, @rh = lh, rh
  end

  def eval(scope)
    @lh.eval(scope) - @rh.eval(scope)
  end
end

class Multiplication < Node
  def initialize(lh, rh)
    @lh, @rh = lh, rh
  end

  def eval(scope)
    @lh.eval(scope) * @rh.eval(scope)
  end
end

class Division < Node
  def initialize(lh, rh)
    @lh, @rh = lh, rh
  end

  def eval(scope)
    @lh.eval(scope) / @rh.eval(scope)
  end
end

class Comparison < Node
  def initialize(lh, op, rh)
    @lh, @rh, @op = lh, rh, op
  end

  def eval(scope)
    lh = @lh.eval(scope)
    rh = @rh.eval(scope)

    lh = "'#{lh}'" if lh.kind_of? String
    rh = "'#{rh}'" if rh.kind_of? String

    Kernel.eval("#{lh} #{@op} #{rh}")
  end
end

class IfStatement < Node
  def initialize(condition, stmt_list, else_stmt_list = nil)
    @condition = condition
    @stmt_list = stmt_list
    @else_stmt_list = else_stmt_list
  end

  def eval(scope)
```

```
      if @condition.eval(scope)
        @stmt_list.eval(scope)
      elsif @else_stmt_list
        @else_stmt_list.eval(scope)
      end
    end
  end

  class WhileStatement < Node
    def initialize(condition, stmt_list)
      @condition, @stmt_list = condition, stmt_list
    end

    def eval(scope)
      while_scope = {"PARENTSCOPE" => scope}
      result = nil
      while @condition.eval(while_scope) and result.class != ReturnStatement do
        result = @stmt_list.eval(while_scope)
      end

      result
    end
  end

  class VariableAssignment < Node
    def initialize(name, node)
      @name = name.to_sym
      @node = node
    end

    def eval(scope)
      value = @node.eval(scope)

      assign(scope, @name, value)

      return value
    end

  end

  class Variable < Node
    def initialize(name)
      @name = name.to_sym
    end

    def eval(scope)
      return look_up(@name, scope)
    end
  end

  class PrintStatement < Node
    def initialize(expression)
      @string = expression
    end

    def eval(scope)
      puts @string.eval(scope)
```

```
    end
  end

  class ReturnStatement < Node
    attr_reader :value

    def initialize(expression)
      @expression = expression
    end

    def eval(scope)
      @value = @expression.eval(scope)
    end
  end

  class FunctionDefinition < Node
    attr_reader :params, :stmt_list, :scope

    def initialize(name, params, stmt_list)
      if name
        @name = name.to_sym
      else
        @name = nil
      end

      @params = params.map { |param| param.to_sym }
      @stmt_list = stmt_list
    end

    def eval(scope)
      @scope = scope
      if @name
        assign(scope, @name, self)
      end

      return self
    end

  end

  class FunctionCall < Node

    def initialize(name, args)
      @name = name.to_sym
      @arguments = args
    end

    def eval(scope)
      # Find function definition in scopes
      function_definition = look_up(@name, scope)

      params = function_definition.params

      if params.length != @arguments.length
        return "Function #{@name.to_s} called with invalid number of \
arguments. (#{@arguments.length} for #{params.length})"
      end
```

```
        # Give our parameters value from the argument-array and create
        # a scope from that.
        evaluated_arguments = @arguments.map { |a| a.eval(scope) }
        function_scope = build_frame(params, evaluated_arguments)
        function_scope["OUTSIDE_FUNCTION_DEF"] = function_definition.scope

        result = nil

        function_definition.stmt_list.each do |statement|

          result = statement.eval(function_scope)

          if statement.kind_of? Dunder::Nodes::ReturnStatement
            result = statement.value
            break
          elsif result.kind_of? Dunder::Nodes::ReturnStatement
            result = result.value
            break
          end

        end

        result
      end

    end

  end
end
```

```ruby
# coding: utf-8

####
# test_helpers.rb
# http://github.com/bjorngylling/TDP019
# Part of the Dunder language project by Björn Gylling and Linus Karlsson
# This file contains helper methods for the tests

require 'lib/dunder.rb'

require 'test/unit'

begin
  require 'rubygems'
  require 'turn'
rescue LoadError
end

class DunderParserTest < Test::Unit::TestCase

  def run_output_test(code)
    # Redirect stdout to output
    output = StringIO.new
    old_stdout, $stdout = $stdout, output

    @d_parser.parse(code).eval(@global_scope)

    # Restore stdout
    $stdout = old_stdout

    output.string
  end

  def parse(code)
    @d_parser.parse(code).eval(@global_scope)
  end

end


class Dunder::Nodes::Node
  def self.sub_classes
    sub_classes = ObjectSpace.enum_for(:each_object, class << self; self; end).to_a
    sub_classes.delete(self)

    sub_classes
  end
end


class DunderNodesTest < Test::Unit::TestCase

  def evaluate(node)
    node.eval(@scope)
  end

  ## CREATE_NODE
```

```ruby
  #
  # Takes in name of the current test method and returns
  # a Node that is supposed to be tested by that method.
  # It takes anything after the last underscore and uses that
  # to determine which Node it should be. Can also be used
  # to create a different node by passing the node name.
  #
  # Tests testing nodes should always be named: test_NODENAME
  # ex. test_addition tests the node Dunder::Node:Addition
  #
  # Any extra arguments passed are passed along to the initialization of the new node
  # such as the value of a int or the terms in an addition node.

  def create_node(test_method, *params)
    # Get available nodes so we can find the one we need, this has to be
    # done since we don't know which letters should be capitalized.
    available_nodes = Dunder::Nodes::Node.sub_classes

    node_name_lower_case = "dunder::nodes::#{get_last_word_in test_method}"
    node_name = available_nodes.select { |node| node.to_s.downcase == node_name_lower_case }.
first

    node_name.new *params
  end

  def get_last_word_in(words_seperated_by_underscore)
    words_seperated_by_underscore.split("_").last
  end

  def node(*params)
    create_node self.name.split("(").first, *params
  end

  def method_missing(name, *args)
    if name.to_s =~ /^int_(\d+)$/
      return create_node "dinteger", $1
    end

    super
  end
end
```

```ruby
# coding: utf-8

####
# nodes_test.rb
# http://github.com/bjorngylling/TDP019
# Part of the Dunder language project by Björn Gylling and Linus Karlsson
# This file contains test-cases testing the nodes

require "test/test_helpers.rb"

class DunderNodesTest < Test::Unit::TestCase

  def setup
    @scope = Hash.new
  end

  def test_addition
    assert_equal 15, evaluate(node(int_10, int_5))
  end

  def test_subtraction
    assert_equal 5, evaluate(node(int_10, int_5))
  end

  def test_multiplication
    assert_equal 50, evaluate(node(int_10, int_5))
  end

  def test_division
    assert_equal 2, evaluate(node(int_10, int_5))
  end

  def test_advanced_addition
    # Create another Addition Node and use it as the left hand
    sub_addition = node int_5, int_6

    assert_equal 21, evaluate(node(sub_addition, int_10))
  end

  def test_dboolean
    assert_equal true, evaluate(node(true))
    assert_equal false, evaluate(node(false))
    assert_equal false, evaluate(node(nil))
  end

  def test_comparison
    assert_equal true, evaluate(node(int_10, "==", int_10))
    assert_equal false, evaluate(node(int_10, "==", int_5))
    assert_equal true, evaluate(node(int_10, "!=", int_5))
    assert_equal false, evaluate(node(int_5, "!=", int_5))
  end

  def test_ifstatement
    stmt_list = create_node("statementlist", create_node("addition", int_5, int_5))
    condition = create_node("dboolean", true)

    assert_equal 10, evaluate(node(condition, stmt_list))
```

```ruby
  end

  def test_variableassignment
    # Make the assignment
    evaluate(node("myVar", int_10))

    assert_equal 10, @scope[:myVar]
  end

  def test_scope_variableassignment
    global_scope = {:another_variable => "yes"}
    @scope["PARENTSCOPE"] = global_scope # Empty variable scope to store our test-variable

    # Make the assignment
    evaluate(node("myVar", int_10))
    assert_equal 10, @scope[:myVar]
    assert_nil global_scope[:myVar] # Should not pollute parent-scope

    # This time it should find "another_variable" in the parent-scope and overwrite it
    evaluate(node("another_variable", int_5))
    assert_equal 5, global_scope[:another_variable]
    assert_nil @scope[:another_variable] # Should not pollute local scope
  end

  def test_variable
    @scope[:myVar] = "hej"

    assert_equal "hej", evaluate(node("myVar"))
  end

  def test_scope_variable
    global_scope = {:another_variable => "yes", :myVar => "goodbye"}
    @scope["PARENTSCOPE"] = global_scope
    @scope[:myVar] = "hej"

    assert_equal "hej", evaluate(node("myVar"))
    assert_equal "yes", evaluate(node("another_variable"))
  end

  def test_statementlist
    # Create a bunch of statements
    stmt_1 = create_node "addition", int_5, int_10
    stmt_2 = create_node "addition", int_5, int_5
    stmt_3 = create_node "multiplication", int_5, int_10

    stmt_list_1 = node stmt_1

    stmt_list_1 += node stmt_2
    assert_equal 2, stmt_list_1.list.length

    stmt_list_1 += node stmt_3
    assert_equal 3, stmt_list_1.list.length
  end

  def test_functioncall
    global_scope = Hash.new
```

```ruby
    # Create a bunch of statements
    stmt_list = Array.new
    stmt_list << create_node("addition", int_5, int_10)
    stmt_list << create_node("addition", int_5, int_5)
    stmt_list << create_node("multiplication", int_5, int_10)

    function = create_node("functiondefinition", "foo", ["a", "b", "c"], stmt_list)
    evaluate function

    assert_equal 50, evaluate(node("foo", [int_12, int_1, int_3]))
  end

end
```

```
# coding: utf-8

####
# parser_test.rb
# http://github.com/bjorngylling/TDP019
# Part of the Dunder language project by Björn Gylling and Linus Karlsson
# This file contains test-cases the language

require "test/test_helpers.rb"

require "stringio"

class DunderParserTest < Test::Unit::TestCase

  def setup
    @global_scope = Hash.new
    @d_parser = nil
    @d_parser = Dunder::Parser.new
  end

  def test_addition
    assert_equal 6, parse("4+2")
  end

  def test_addition_with_strings
    assert_equal "hej då", parse("'hej' + ' då'")
  end

  def test_subtraction
    assert_equal 6, parse("8-2")
  end

  def test_multiplication
    assert_equal 16, parse("8*2")
  end

  def test_division
    assert_equal 4, parse("8/2")
  end

  def test_parser_function_remove_unwanted_newlines_in
    string = "Here comes two empty rows with tabs


and here is the last row"
    expected_result = "Here comes two empty rows with tabs
and here is the last row"
    assert_equal expected_result, @d_parser.remove_unwanted_newlines_in(string)
  end

  def test_whitespace
    assert_equal 6, parse("4 +2")
    assert_equal 6, parse("4    + 2")
    assert_equal 6, parse("4 +2\n")
    assert_equal 6, parse("#HEEEJ\n\n\n4 +2\n")
  end
```

```
  def test_statement_list
    assert_equal 35, parse("1*4;33 + 2")
    assert_equal 6, parse("1*4\n4 + 2")
  end

  def test_math_priority
    assert_equal 7, parse("1+2*3")
    assert_equal -1, parse("1-5+3")
    assert_equal 7, parse("3*2+1")
    assert_equal 8, parse("1+3*2+1")
    assert_equal -1, parse("1-10/2+3")
    assert_equal 48, parse("1+10*10/2-3")
  end

  def test_negativity
    assert_equal 4, parse("1--3")
    assert_equal 10, parse("-2*-5")
    assert_equal 1, parse("-4+5")
    assert_equal -9, parse("-4+-5")
    assert_equal -2.4, parse("-4.5--2.1")
  end

  def test_zero_fill
    assert_equal 12, parse("00001*0001+0000000011")
    assert_equal 12.1, parse("00001*0001+0000000011.10")
  end

  def test_math_with_variables
    code = "var = 100 * 10
            x = 12
            var = var + x"
    assert_equal 1012, parse(code)
  end

  def test_math_with_float
    assert_equal 5.3, parse("10.5 - 5.2")
    assert_equal 5.1, parse("10.2 * 0.5")
    assert_equal 5.1, parse("10.2 * .5")
  end

  def test_boolean
    assert_equal true, parse("true")
    assert_equal false, parse("false")
  end

  def test_comparison
    assert_equal true, parse("10 == 10")
    assert_equal true, parse("10 != 9")
    assert_equal true, parse("'hej' == 'hej'")
    assert_equal true, parse("100 > 99")
    assert_equal true, parse("var = 12;result = var < 42; result")
    assert_equal true, parse("100 >= 100")
    assert_equal true, parse("100.00 == 100")
    assert_equal false, parse("100 > 100")
    assert_equal true, parse("100 >= 99")
    assert_equal true, parse("100 != 'aaa'")
  end
```

```
def test_ifstatement
  assert_equal 102, parse("if(10 == 10) {
                             100 + 2
                           }")
  assert_equal nil, parse("if(10 == 10) {
                             if(20==19) { 100+15 }
                           }")
  assert_equal 115, parse("if(10 == 10) {
                             if(20==20) { 100+15 }
                           }")
  assert_equal false, parse("if(10 == 11) {
                               arne = 20
                             }
                             arne")
end

def test_ifelsestatement
  assert_equal 102, parse("if(10 == 10) {
                             100 + 2
                           } else {
                             100 - 2
                           }")

  assert_equal 98, parse("if(10 == 11) {
                            100 + 2
                          } else {
                            100 - 2
                          }")

  assert_equal "hej", parse("if(10 == 11) {
                               arne = 20
                             } else {
                               foo = 'hej'
                             }
                             foo")

  code = "x = 12
          y = 10 + 2
          if(y == x) {
            new_var = x + 32
          } else {
            new_var = y + 30
          }
          new_var"
  assert_equal 44, parse(code)
end

def test_whilestatement
  assert_equal 10, parse("x = 1;
                          while(x < 10) { x = x + 1 }; x")

  code = "x = 1
          answer = 1
          number = 5
          while(x <= number) { #Körs tills x är större än number
            #Multiplicerar det nuvarande numret med gamla faktorn
```

```
            answer = answer * x
            x = x + 1
          }
          answer"
  assert_equal 120, parse(code)

end

def test_variable_assignment
  assert_equal "hej", parse("var = 'hej'")
  assert_equal 100, parse("var = 100")
  assert_equal 1000, parse("var = 100 * 10")
  assert_equal 20, parse("x = 19; x = x + 1")
end

def test_string_with_nonstandard_characters
  assert_equal "heh&&lll", parse('var = "heh&&lll"')
end

def test_variable_reading
  assert_equal 5, parse("var = 5")
  assert_equal 5, parse("var")
  assert_equal 200, parse("var = 100 * 2")
  assert_equal 200, parse("var")


  assert_equal 1000, parse("var = 100 * 10; var")
  assert_equal 1000, parse("var = 100 * 10;12 + 2; var")
  assert_equal 1000, parse("var = 100 * 10
                            12 + 2
                            var")
  assert_equal 1000, parse("var = 100 * 10;x = 12; var")
end

def test_scope
  code = "x = 0
          while(x < 5) {
            x = x + 1
            z = x
          }"

  parse(code)

  assert_equal false, parse("z") # z should not be set
  assert_equal 5, parse("x")
end

def test_comments
  assert_equal 1000, parse("var = 100 * 10
                            12 + 2 # var = 10
                            var")
  assert_equal 1000, parse("var = 100 * 10
                            12 + 2 // var = 10
                            var")
  assert_equal 1000, parse("var = 100 * 10
                            12 + 2 # var = 12
                            /* hejeje */ comment does not end here
```

```
                                var = 100
                               dfd*/ var")
  end

  def test_function_definition
    assert parse("def foo() { 10 }")
  end

  def test_function_without_arguments
    parse("def foo() { 10 }")

    assert_equal 10, parse("foo()")
  end

  def test_function_with_arguments
    parse("def foo(x) { x + 10 }")

    assert_equal 20, parse("foo(10)")
  end

  def test_function_with_multiple_arguments
    parse("def foo(x, y) { x + y }")

    assert_equal 15, parse("foo(10, 5)")
  end

  def test_function_with_variable_as_argument

    parse("def foo(x) { x + 10 }")

    assert_equal 40, parse("var = 30; foo(var)")
  end

  def test_function_with_return
    parse("def foo(x) { x + 10
             // Ugly formatting to make sure that works too
             return 12
             13 }")
    assert_equal 12, parse("foo(10)")

    parse("def foo(x) {
             x = x + 10
             return x + 5
             13
           }")
    assert_equal 25, parse("foo(10)")

    parse("def foo(x) {
             x = x + 10
             if(x > 2) {
               return x
             }
             13
           }")
    assert_equal 20, parse("foo(10)")

    parse("def foo(y) {
```

```
            z = 0
            while(z < 20) {
              if(z == y) {
                return z * 10
              }
              z = z + 1
            }
            return 42
          }")
    assert_equal 100, parse("foo(10)")
  end

  def test_recursive_functions
    code = "def foo(x) {
              if(x >= 100) {
                return x
              }
              else {
                return foo(x + 10)
              }
            }"

    parse(code)

    assert_equal 100, parse("foo(0)")

    code = "def fib(n) {
              if (n <= 1) {
                return n
              } else {
                return fib(n-1)+fib(n-2)
              }
            }"
    parse(code)

    assert_equal 8, parse("fib(6)")
  end

  def test_lambda_functions
    code = "foo = { |var| var = var + 10 }
            foo(22)"

    assert_equal 32, parse(code)
  end

  def test_run_code_from_file
    dir = $:.unshift File.dirname(__FILE__)
    result = %x[ruby lib/dunder.rb test/fixtures/factorial.dun]
    assert_equal 120, result.chomp.to_i
  end

  def test_print_string
    code = "x = 0
            while(x < 5) {
              x = x + 1
              print 'Hello world'
            }"
```

```
    assert_equal ("Hello world\n" * 5), run_output_test(code)
  end

  def test_print_number
    code = "x = 0
            while(x < 4) {
              x = x + 1
              print 100 - 50
            }"

    assert_equal ("50\n" * 4), run_output_test(code)
  end

  def test_static_binding
    dir = $:.unshift File.dirname(__FILE__)
    result = %x[ruby lib/dunder.rb test/fixtures/static_binding.dun]
    assert_equal 10, result.chomp.to_i

    result = %x[ruby lib/dunder.rb test/fixtures/static_binding_with_lambda.dun]
    assert_equal 15, result.chomp.to_i
  end

end
```