Användar- och systemdokumentation

Linköpings universitet
TDP019 Projekt: Datorspråk
Examinator: Anders Haraldsson, IDA

Martin Melin
marme647@student.liu.se
Utskriftsdatum: 2011-05-31

Innehållsförteckning

Inledning

Paxl skapades våren 2011 av Martin Melin under projektkursen TDP019 Datorspråk på IP-programmet. Grundidén bakom Paxl är att kombinera den kalkylbladsmodell som bland annat Microsoft Excel använder med ett generellt programmeringsspråk. Syftet är att underlätta programmeringen av parallella datorprogram genom att organisera programmets källkod i rader och kolumner. Jag fick idén från den bild som finns på första sidan av *"Introduction to Parallel Programming and MapReduce", Google Code University.*

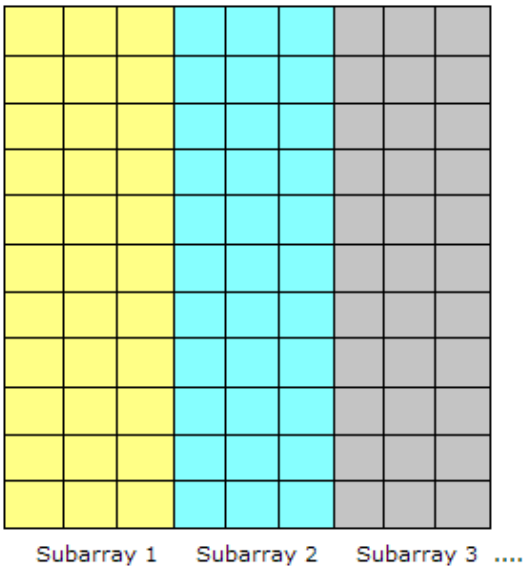

Bild 1: Uppdelning av indata i delmängder

Bilden visar hur den parallella programmeringsmodellen MapReduce delar upp indata i kolumner som sedan bearbetas oberoende av varandra. Jag slogs av att bilden liknande ett kalkylblad med rader och kolumner av data. Min tanke blev då att det borde vara lätt att förklara parallell bearbetning av data genom att översätta det till ett välkänt program som Microsoft Excel.

Tanken med Paxl är således att göra parallell bearbetning av data tillgänglig för användare av Microsoft Excel och andra kalkylprogram. Genom att ha explicita begränsningar av räckvidd, som förklaras närmare längre in i denna rapport, så blir parallell programmering tillgänglig även för den som inte har någon tidigare erfarenhet av teorin bakom parallellism.

Användarhandledning

Det är viktigt att förstå att Paxl består av två olika delar, dels ett datorspråk, med egen syntax som förklaras här nedan, dels den tolk som står för in- och utmatning av Excelfiler och parallell exekvering av den kod som finns däri. Den här handledningen fokuserar först på språket Paxl och dess syntax, för att avslutas med en kortare demonstration av tolken Paxl.

Systemkrav och installation

För att kunna använda språket Paxl behöver du en fungerande Ruby-installation, version 1.8.7 eller senare. För att kunna använda tolken Paxl behöver du kunna skapa filer av typ .xls (den klassiska Excelversionen). Utveckling har skett med OpenOffice.org men Microsoft Excel fungerar också. För att kunna använda Paxl-tolken behöver du installera biblioteket *spreadsheet* med Rubygems.

Paxl har en interaktiv tolk som främst är tänkt att användas för att snabbt kunna testa språket. Starta den interaktiva tolken med:

```
$ ruby paxl_interactive.rb
Welcome to the Paxl interactive parser (type
'exit' to quit)
Paxl: 1+1
=> 2
Paxl:
```

Eftersom tolken kör varje uttryck efter en radbrytning så rekommenderas semikolon som separator för flera uttryck på samma rad:

```
Paxl: a = 1 + 1; b = 3; a * b
=> 6
```

För att avsluta tolken, skriv exit:

```
Paxl: exit
Goodbye!
$
```

Språkets syntax

Paxl bryr sig inte om blanka tecken annat än tecknet för ny rad som kan användas för att avsluta ett uttryck. Man kan skriva flera uttryck på samma rad i Paxl genom att avsluta uttrycken med semikolon istället för ny rad. Språket är skiftlägeskänsligt och alla inbyggda identifierare är i gemener.

Block

Paxl behandlar kod som annan data. Block kan sparas i variabler och skickas som argument till andra block. När ett block definieras så sparas omgivningen, en s.k. closure.

```
x = 5; y = 3
multiply = { | a, b | x * y * a * b }
x = 1; multiply(10; x)
```

Resultatet av det sista uttrycket ovan blir alltså 5 * 3 * 10 * 1 = 150. Notera att argument är uttryck och skiljs åt med semikolon på samma sätt som andra uttryck.

Det finns ett reserverat nyckelord, `this`, som refererar till det omkringliggande blocket. Detta möjliggör rekursion på ett enkelt sätt:

```
factorial = { |x|
     if (x == 0) { 1 }
     else { x * this(x - 1) }
}
```

Block får sin omgivning vid definitionstillfället och har därmed inte sitt eget namn tillgängligt, därför måste man anropa `this` istället för `factorial` i koden ovan. Detta är en följd av att Paxl tillåter anonyma block, alltså block som inte har något namn tilldelat. Eftersom man i det fallet ändå behöver nyckelordet `this` för att kunna göra rekursiva anrop så blir det en mer generell lösning att använda `this` för alla block.

```
x = 5
f = { |a| a + x }
g = { |x| f(x) }
n = g(10)
```

Detta exempel illustrerar Paxls statiska bindning. Här får alltså variabeln `n` värdet `15` och inte `20` som resultatet hade blivit med dynamisk bindning.

If-satser

If- och if/else-satser har ett föga förvånande utseende i Paxl.

```
if ( x < 10 ) { x = 10 } else { x = 20 }
```

For-loopar

Inte heller for-loopar har en syntax som bjuder på några större överraskningar.

```
for ( x = 1; x < 10; x = x + 1) {
     do_something_interesting(x)
}
```

Datatyper

Paxl har endast tre inbyggda datatyper: *sanningsvärden*, *heltal* och *flyttal*.

```
x = true; y = 10; z = 3.14
```

Logiska uttryck

Paxl har de logiska operatorerna and, or och not. Andra uttryck kan användas i ett logiskt uttryck, där värdet på uttrycket tolkas som ett sanningsvärde.

```
x = (true and false) or true
y = 10
z = (y < 10) or something_else(x) or true
```

Aritmetiska uttryck

De fyra räknesätten finns representerade i Paxl och fungerar som vanligt.

```
x = 10 + 5 - 8
y = 10 * 5 / 8
z = 10 + 5 * 8
```

Jämförelser

Jämförelseoperatorer kan användas i alla uttryck och returnerar sanningsvärden.

```
x = 10 <= 5
y = 5 == 5
z = x != y
```

Tolken

Paxl-tolken fungerar genom att läsa .xls-filer och tolka varje cells innehåll som ett kodblock. Varje cells räckvidd är begränsad till returvärden för alla celler som ligger till vänster om den nuvarande på samma rad. Ett exempel gör detta tydligare. De översta raderna tillhör inte filen utan visar kolumnernas etiketter.

A	B	C	D
15	20	`if (A >= 10) { 10 } else { A }`	`C * D`
5	6		
3	2		

När tolken körs med en .xls-fil med ovanstående innehåll så skapar den en ny fil med lika många rader och kolumner, där varje cell har bytts ut till returvärdet från koden som körts.

A	B	C	D
15	20	10	200
5	6	5	30
3	2	3	6

Grundkonfigurationen av tolken startar fyra trådar och kör alla rader i dokumentet jämnt fördelat över dem.

Kör tolken genom att anropa den med input-filens filnamn:

```
$ ruby paxl_xls.rb test.xls
```

Systemdokumentation

Paxl är byggt i programspråket Ruby med hjälp av rdparse som också använts i kursen TDP007 Datorspråk. Dessutom används biblioteket spreadsheet för läsning och skrivning av .xls-filer, samt Test::Unit för enhetstestning.

Lexikalisk analys

Tack vare användningen av rdparse så består den lexikaliska analysen i Paxl endast av ett par reguljära uttryck som konfigurerar rdparserns tokenisering:

```
token(/\s+/)
token(/and|or|==|!=|<=|>=|\w+|./) { |m| m }
```

Ovanstående rader gör så att alla blanka tecken kastas bort, därefter tolkas alla sammanhängande ord som en egen token, och ett antal speciella operatorer. På detta sätt hålls den lexikaliska analysdelen till ett minimum.

Parsning

Parsningsreglerna matchar olika tokens och skapar objekt av olika klasser, s.k. noder, i en trädstruktur som representerar parserns tolkning av programmet ifråga. Det färdiga trädet returneras sedan, där toppnoden alltid är av typen PStatementList. Paxl gör med andra ord ingen åtskillnad på början/slut av program och andra listor av uttryck. Detta generella tillvägagångssätt har underlättat vid utvecklingsarbetet.

Evaluering

Varje klass av nod har en egendefinierad eval-funktion som bestämmer vad som ska göras för att returnera ett värde. Alla noder i Paxl returnerar ett värde, förutom PBlockDefinition som klonar den omkringliggande omgivningen och därefter returnerar sig själv. Det möjliggör hantering av kod som data, där man kan sätta en variabel till ett block för att senare evaluera blocket.

Omgivningshantering

Paxl hanterar omgivningar med objekt av klassen Scope, som ärver från den inbyggda Ruby-klassen Hash. Det innebär att man kan använda Scope-objekt väldigt enkelt, t.ex. så här:

```
my_scope["a"] = 5
```

Vilket sätter variabeln med namn a till värdet 5. Det intressanta med Scope är att den stödjer hierarkiskt ordnade omgivningar, med tydlig logik för räckvidd. Koden som implementerar detta i Ruby ser ut som följer:

```
def [](key)
  if self.has_key? key
    return super(key)
  else
    if @parent.nil?
      return nil
    else
      return @parent[key]
    end
  end
end
def []=(key, value)
  current_value = self[key]
  if self.has_key? key or current_value.nil?
    super(key, value)
  else
    @parent[key] = value
  end
end
```

Detta innebär i klartext att underliggande Scope-objekt har tillgång till alla värden från ovanliggande Scope, och vid konflikter är det alltid den variabel som är definierad närmast nuvarande Scope som vinner. På samma sätt kan kod ändra i variabler från omkringliggande Scope, men inte sätta nya variabler i Scope ovanför sitt eget. Nya Scope-objekt skapas t.ex. vid blockdefinitioner vilket är det som möjliggör statisk bindning.

Nodtyper

Paxl har 16 olika klasser av noder.

PStatementList

Innehåller en array med en eller flera andra noder. Evaluering sker genom att anropa eval för varje element i arrayen i ordning och därefter returnera det senaste.

PForLoop

Fungerar som en traditionell for-loop i t.ex. C. Antar att det finns tre uttryck i huvudet och evaluerar blocket i kroppen tills det mittersta uttrycket ger ett falskt returvärde.

PIfStatement

Accepterar vilket uttryck som helst i huvudet, tolkar returvärdet som ett sanningsvärde och kör därefter motsvarande block.

PBlockDefinition

Den enda nod som istället för ett vanligt värde returnerar sig själv vid evaluering. Klonar den omgivning som finns vid evaluering och sparar denna i sig själv, en s.k. closure.

PBlockCall

Anrop av block. Först vid evaluering kontrolleras att det värde som finns i variabeln som anropas faktiskt är ett objekt av klassen PBlockDefinition. Argument kan vara vilka uttryck som helst.

PVariableAssignment

Tilldelning av värde till en variabel i nuvarande omgivning.

PVariableReference

Hämtning av variabels värde från nuvarande omgivning.

PLogicalExpression

Ett logiskt uttryck med operatorerna AND, OR samt NOT. Tolkar sina argument som sanningsvärden.

PComparison

Jämför två uttrycks returvärden enligt de vanliga jämförelseoperatorerna och enligt Rubys logik för jämförelse. Medger t.ex. att 'A' <= 'Z'.

PMultiplication

Vanlig multiplikation av två uttrycks returvärden.

PDivision

Vanlig division av två uttrycks returvärden.

PAddition

Vanlig addition av två uttrycks returvärden.

PSubtraction

Vanlig subtraktion av två uttrycks returvärden.

PInteger

Representerar ett heltal.

PFloat

Representerar ett flyttal.

PBoolean

Representerar ett booleskt sanningsvärde.

Grammatik

```
<stmt_list>      ::=   <stmt>
                       | <stmt_list> \n <stmt>
                       | <stmt_list> ; <stmt>

<stmt>           ::=   <if_stmt>
                       | <for_loop>
                       | <expr>

<expr>           ::=   <block_def>
                       | <block_call>
                       | <expr> + <expr>
                       | <expr> - <expr>
                       | <expr> * <expr>
                       | <expr> / <expr>
                       | <cond_stmt>
                       | <var_assign>

<block_call>     ::=   <identifier> ()
                       | <identifier> ( <stmt_list> )

<block_def>      ::=   { | <param_list> | <stmt_list> }
                       | { <stmt_list> }

<if_stmt>        ::=   if ( <expr> )
                          { <stmt_list> }
                       | if ( <expr> )
                            { <stmt_list> }
                            else
                            { <stmt_list> }

<for_loop>       ::=   for ( <expr> ; <expr> ; <expr> )
                           { <stmt_list> }

<cond_stmt>      ::=   <expr> <cond_op> <expr>

<cond_op>        ::=   ==
                       | !=
                       | <=
                       | >=
                       | <
                       | >

<var_assign>     ::=   <identifier> = <expr>

<identifier>     ::=   [a-zA-Z][a-zA-Z0-9_]*
```

Reflektioner

Det har varit ett mycket intressant och lärorikt projekt att utföra. Den främsta utmaningen för mig har varit att försöka översätta min grundidé till något som kan utföras inom ramen för detta projekt. Jag är nöjd med resultatet av detta projekt.

Det är en intressant motsättning i programspråkskonstruktion mellan att göra något enkelt för nybörjare att använda och att göra något som inte går vana användare på nerverna.

Vilken syntax man väljer kan i stor utsträckning bestämmas av målgrupp och dess erfarenhet av programmering. Är målgruppen totala nybörjare så bör man t.ex. inte använda för många specialtecken utan det är bättre att ha ett nyckelordsstyrt språk (exv Basic). Om man riktar sig till vanliga programmerare så kan det finnas ett värde i att inte överraska för mycket, alltså att språkets syntax liknar det som målgruppen är van vid (exv att Javascript i sista stund fick en C-liknande syntax påklistrad av marknadsföringsskäl). Och slutligen om man riktar sig till mycket intresserade och kunniga programmerare så kan man verkligen ta ut svängarna lite mer, eftersom målgruppen då redan är van vid att ta till sig ny syntax (exv Haskell, Lisp, Erlang).

Ett exempel är att jag personligen föredrar indenteringsstyrda språk, av den enkla anledningen att man ändå bör indentera sin kod på ett konsekvent sätt och då är det onödigt att ha extra tecken för något som redan är implicit av indenteringen. Men i utvecklingen av Paxl så valde jag att inte göra det indenteringsstyrt, eftersom jag märkt att det skapar svårighet och frustration för nya användare som inte är vana vid att bry sig om blanka tecken.

```ruby
#!/usr/bin/env ruby

require 'lib/rdparse'
require 'nodes'

module Paxl

  class Parser

    include Paxl::Nodes

    def initialize(log = false)

      @parser = Rdparse::Parser.new('paxl', log) do

        @scope = Hash.new

        token(/\s+/)
        token(/and|or|==|!=|<=|>=|\w+|./) { |m| m }

        start :stmt_list do
          match(:stmt, :stmt_term, :stmt_list) { |a, _, b| b = a + b }
          match(:stmt, :stmt_term)
          match(:stmt)
        end

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

        rule :stmt do
          match(:if_stmt) { |a| PStatementList.new a }
          match(:for_loop) { |a| PStatementList.new a }
          match(:expr) { |a| PStatementList.new a }
        end

        rule :for_loop do
          match('for', '(', :stmt_list, ')', '{', :stmt_list, '}') do
            |_, _, a, _, _, b, _| PForLoop.new(a, b)
          end
        end

        rule :if_stmt do
          match('if', '(', :stmt_list, ')', '{', :stmt_list, '}',
            'else', '{', :stmt_list, '}') do
            |_, _, a, _, _, b, _, _, _, c, _| PIfStatement.new(a, b, c)
          end
          match('if', '(', :stmt_list, ')', '{', :stmt_list, '}') do
            |_, _, a, _, _, b, _| PIfStatement.new(a, b, nil)
          end
        end

        rule :expr do
          match('{', '|', :param_list, '|', :stmt_list, '}') do
            |_, _, a, _, b, _| PBlockDefinition.new(a, b)
          end
          match('{', :stmt_list, '}') { |_, a, _| PBlockDefinition.new([], a) }
          match(:identifier, '(', ')') { |a, _, _| PBlockCall.new(a, []) }
          match(:identifier, '(', :stmt_list, ')') { |a, _, b, _| PBlockCall.new(a, b) }
          match(:expr, '+', :term) { |a, _, b| PAddition.new(a, b) }
```



```ruby
          match(:expr, '-', :term) { |a, _, b| PSubtraction.new(a, b) }
          match(:expr, :logical_operator, :term) { |a, op, b| PLogicalExpression.new(a,
          match('not', :expr) { |op, a| PLogicalExpression.new(a, op, PBoolean.new(1)) }
          match(:expr, :comparison_operator, :term) do
            |a, op, b| PComparison.new(a, op, b)
          end
          match(:term)
        end

        rule :term do
          match(:term, '*', :atom) { |a, _, b| PMultiplication.new(a, b) }
          match(:term, '/', :atom) { |a, _, b| PDivision.new(a, b) }
          match(:atom)
        end

        rule :var do
          match(:identifier, '=', :expr) { |a, _, b| PVariableAssignment.new(a, b) }
          match(:identifier) { |a| PVariableReference.new(a) }
        end

        rule :param_list do
          match(:identifier, ',', :param_list) { |a, _, b| b + [a] }
          match(:identifier) { |a| [a] }
        end

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

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

        rule :logical_operator do
          match('and')
          match('or')
        end

        rule :atom do
          match(:boolean)
          match(:number)
          match(:var)
          match('(', :expr, ')') { |_, a, _| a }
        end

        rule :boolean do
          match('true') { PBoolean.new(1) }
          match('false') { PBoolean.new(0) }
        end

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

        rule :float do
```

```
123           match(:digits, '.', :digits) { |a, _, b| PFloat.new("#{a}.#{b}") }
124           match('.', :digits) { |_, a| PFloat.new("0.#{a}") }
125         end
126
127         rule :integer do
128           match(:digits) { |a| PInteger.new(a) }
129         end
130
131         rule :digits do
132           match(:digits, :digit) { |a, b| a += b }
133           match(:digit)
134         end
135
136         rule :digit do
137           match(/[0-9]/)
138         end
139
140       end
141
142     end
143
144     def parse(code)
145       @parser.parse(code)
146     end
147
148     def interactive
149       puts "Welcome to the Paxl interactive parser (type 'exit' to quit)"
150       code = ""
151       global_scope = Paxl::Scope.new(nil)
152       while true
153         print "Paxl: "
154         line = gets
155         break if line.chomp == "exit"
156
157         code += line
158
159         result = parse(code).eval(global_scope)
160         print "=> "
161         puts result
162         code = ""
163       end
164       puts "Goodbye!"
165     end
166
167   end
168
169 end
```

```ruby
#!/usr/bin/env ruby
module Paxl
  module Nodes

    class Node
    end

    class PStatementList < 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)
        end
        return result
      end

    end

    class PForLoop < Node

      def initialize(control, statements)
        @control, @statements = control, statements
      end

      def eval(scope)
        my_scope = Paxl::Scope.new(scope)
        init_stmt, test_stmt, iter_stmt = @control.list
        init_stmt.eval(my_scope)
        return_value = nil
        while test_stmt.eval(my_scope) do
          return_value = @statements.eval(my_scope)
          iter_stmt.eval(my_scope)
        end
        return_value
      end

    end

    class PIfStatement < Node

      def initialize(test, if_true, if_false)
        @test, @if_true, @if_false = test, if_true, if_false
      end

```



```ruby
      def eval(scope)
        test_result = @test.eval(scope)
        if test_result
          @if_true.eval(scope)
        else
          if @if_false
            @if_false.eval(scope)
          else
            nil
          end
        end
      end

    end

    class PBlockDefinition < Node

      attr_reader :parameters, :statement_list, :block_scope

      def initialize(parameters, statements)
        @parameters, @statement_list = parameters, PStatementList.new(statements)
      end

      def eval(scope)
        @block_scope = scope.clone
        @block_scope["this"] = self
        self
      end

    end

    class PBlockCall < Node

      def initialize(identifier, arguments)
        @identifier, @arguments = identifier, arguments
      end

      def eval(scope)
        block_def = scope[@identifier]
        if not block_def.kind_of? Paxl::Nodes::PBlockDefinition
          return "Error! Attempting to use a non-block as a block."
        end
        my_scope = block_def.block_scope.clone
        if @arguments.kind_of? Paxl::Nodes::PStatementList
          arguments = Array.new(@arguments.list)
        end
        block_def.parameters.each do |param|
          my_scope[param] = arguments.pop().eval(scope)
        end
        block_def.statement_list.eval(my_scope)
      end

    end

    class PVariableAssignment < Node

      def initialize(identifier, value)
        @identifier, @value = identifier, value
      end

      def eval(scope)
```

```
require 'test/unit'
require 'paxl'

class TestPaxl < Test::Unit::TestCase

  # all tests run using the same global scope
  def initialize(arg)
    super(arg)
    @paxl = Paxl::Parser.new
    @paxl_scope = Paxl::Scope.new(nil)
  end

  def assert_returns(value, code)
    assert_equal(value, @paxl.parse(code).eval(@paxl_scope))
  end

  def test_y_combinator
    assert_returns 120, "
    y = { |g|
      f = { |x|
        { |arg|
          gx = g(x(x));
          gx(arg)
        }
      };
      f(g)
    };
    fg = { |cb|
      { |arg|
        if (arg == 0) { 1 }
        else { arg * (cb(arg - 1)) }
      }
    };
    factorial = y(fg);
    factorial(5)"
  end

  def test_this_keyword
    assert_returns 120, "
      factorial = { |x|
        if (x == 0) { 1 }
        else { x * (this(x - 1)) }
      };
      factorial(5)"
  end

  def test_for_loops
    assert_returns 45, "a = 0; for (i = 0; i < 10; i = i + 1) { a = a + i }; a"
  end

  def test_if_statements
    assert_returns 10, "a = 5; if (a < 10) { 10 } else { a }"
    assert_returns 15, "a = 15; if (a < 10) { 10 } else { a }"
    assert_returns 10, "a = 10; if (a == 10) { 10 }"
  end

  def test_blocks
    assert_returns 200, "a = { |b, c| b * c }; a(10; 20)"
    assert_returns 250, "n = 50; a = { |b, c| b * c + n }; a(10; 20)"
    assert_returns 10, "a = { 10 }; a();"
    assert_returns 10, "a = { |x, y, z| (x * y) - z }; a(5; 4; 10)"
```



```
  end

  def test_variable_assignment
    assert_returns 10, "a = 10"
    assert_returns 10, "a = 10"
    assert_returns 5, "b = 5"
    assert_returns 10, "a"
    assert_returns 5, "b"
    assert_returns 50, "a * b"
  end

  def test_basic_math
    assert_returns 10, "5 + 5"
    assert_returns 10, "15 - 5"
    assert_returns 10, "100 / 10"
    assert_returns 10, "5 * 2"
  end

  def test_logical_expressions
    assert_returns true, "true"
    assert_returns false, "false"
    assert_returns true, "true and true"
    assert_returns false, "true and false"
    assert_returns true, "false or true"
    assert_returns true, "not false"
  end

  def test_comparisons
    assert_returns true, "10 == 10"
    assert_returns false, "1 == 2"
    assert_returns true, "1 != 2"
    assert_returns true, "1 < 2"
    assert_returns true, "1 <= 2"
    assert_returns false, "1 >= 2"
    assert_returns false, "1 > 2"
  end

  def test_nested_expressions
    assert_returns true, "(true or false) and (true and true)"
    assert_returns true, "(10 < 20) and (20 >= 10)"
    assert_returns true, "a = ( (true or false) and (true and true) )"
  end

end
```

```ruby
#!/usr/bin/env ruby
require 'rubygems'
require 'spreadsheet'
require 'paxl'

filename = ARGV.shift
if not filename
  abort "I expect a .xls filename as an argument"
end
book = Spreadsheet.open filename
sheet = book.worksheet 0
number_of_threads = 4
threads = []

sheet.each_slice(number_of_threads) do |rows|
  threads << Thread.new(rows) do |rows|
    p = Paxl::Parser.new
    rows.each do |row|
      row_scope = Paxl::Scope.new(nil)
      column_labels = ('A'..'Z').to_a
      values = []
      row.each do |cell|
        code = "#{cell}"
        lbl = column_labels.shift
        result = p.parse(code).eval(row_scope)
        row_scope[lbl] = result
        values << result
      end
      puts "#{row.idx}," + values.join(",")
    end
  end
end

threads.each { |t| t.join }
```

```
123         scope[@identifier] = @value.eval(scope)
124       end
125
126     end
127
128     class PVariableReference < Node
129
130       def initialize(identifier)
131         @identifier = identifier
132       end
133
134       def eval(scope)
135         scope[@identifier]
136       end
137
138     end
139
140     class PLogicalExpression < Node
141
142       def initialize(a, op, b)
143         @a, @op, @b = a, op, b
144       end
145
146       def eval(scope)
147         a, b = @a.eval(scope), @b.eval(scope)
148         case @op
149         when "and"
150           return PBoolean.new( (a and b) ).eval(scope)
151         when "or"
152           return PBoolean.new( (a or b) ).eval(scope)
153         when "not"
154           return PBoolean.new( (not a) ).eval(scope)
155         end
156       end
157
158     end
159
160     class PComparison < Node
161
162       def initialize(a, op, b)
163         @a, @op, @b = a, op, b
164       end
165
166       def eval(scope)
167         a, b = @a.eval(scope), @b.eval(scope)
168         case @op
169         when "=="
170           return PBoolean.new( (a == b) ).eval(scope)
171         when "!="
172           return PBoolean.new( (a != b) ).eval(scope)
173         when "<="
174           return PBoolean.new( (a <= b) ).eval(scope)
175         when ">="
176           return PBoolean.new( (a >= b) ).eval(scope)
177         when "<"
178           return PBoolean.new( (a < b) ).eval(scope)
179         when ">"
180           return PBoolean.new( (a > b) ).eval(scope)
181         end
182       end
183
```



```
184     end
185
186     class PMultiplication < Node
187
188       def initialize(a, b)
189         @a, @b = a, b
190       end
191
192       def eval(scope)
193         @a.eval(scope) * @b.eval(scope)
194       end
195
196     end
197
198     class PDivision < Node
199
200       def initialize(a, b)
201         @a, @b = a, b
202       end
203
204       def eval(scope)
205         @a.eval(scope) / @b.eval(scope)
206       end
207
208     end
209
210     class PAddition < Node
211
212       def initialize(a, b)
213         @a, @b = a, b
214       end
215
216       def eval(scope)
217         @a.eval(scope) + @b.eval(scope)
218       end
219
220     end
221
222     class PSubtraction < Node
223
224       def initialize(a, b)
225         @a, @b = a, b
226       end
227
228       def eval(scope)
229         @a.eval(scope) - @b.eval(scope)
230       end
231
232     end
233
234     class PInteger < Node
235
236       def initialize(value)
237         @value = value.to_i
238       end
239
240       def eval(scope)
241         @value
242       end
243
244     end
```

```
245
246     class PFloat < Node
247
248       def initialize(value)
249         @value = value.to_f
250       end
251
252       def eval(scope)
253         @value
254       end
255
256     end
257
258     class PBoolean < Node
259
260       def initialize(value)
261         if value == true or value == false
262           @value = value
263         else
264           @value = (value != 0) # false if value == 0, true otherwise
265         end
266       end
267
268       def eval(scope)
269         @value
270       end
271
272     end
273
274   end
275
276   class Scope < Hash
277     def initialize(parent)
278       super
279       @parent = parent
280     end
281
282     def [](key)
283       if self.has_key? key
284         return super(key)
285       else
286         if @parent.nil?
287           return nil
288         else
289           return @parent[key]
290         end
291       end
292     end
293
294     def []=(key, value)
295       current_value = self[key]
296       if self.has_key? key or current_value.nil?
297         super(key, value)
298       else
299         @parent[key] = value
300       end
301     end
302
303   end
304 end
```