



## Användar- och systemdokumentation

## Innehållsförteckning

Inledning.....	3
Användarhandledning.....	4
Systemkrav och installation.....	4
Språkets syntax.....	5
Block.....	5
If-satser.....	6
For-loopar.....	6
Datatyper.....	6
Logiska uttryck.....	6
Aritmetiska uttryck.....	6
Jämförelser.....	6
Tolken.....	7
Systemdokumentation.....	8
Lexikalisk analys.....	8
Parsning.....	8
Evaluering.....	8
Omgivningshantering.....	9
Nodtyper.....	10
PStatementList.....	10
PForLoop.....	10
PIfStatement.....	10
PBlockDefinition.....	10
PBlockCall.....	10
PVariateAssignment.....	10
PVariateReference.....	10
PLogicalExpression.....	11
PComparison.....	11
PMultiplication.....	11
PDivision.....	11
PAddition.....	11
PSubtraction.....	11
PInteger.....	11
PFloat.....	11
PBoolean.....	11
Grammatik.....	12
Reflektioner.....	13

## 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.

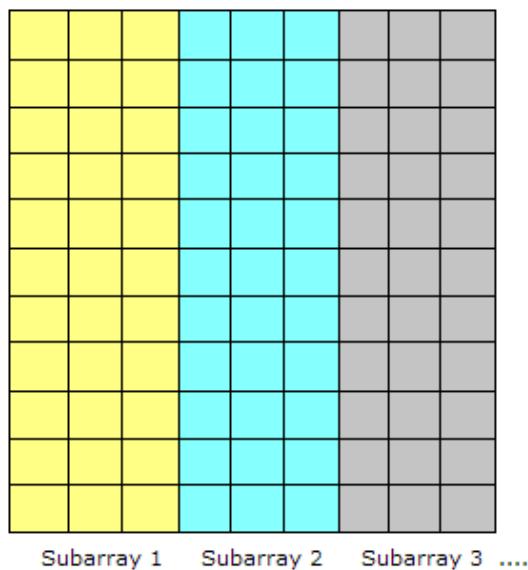


Bild 1: Uppdelning av indata i delmängder

Bilden visar hur den parallella programmeringsmodellen MapReduce delar upp indata i kolumner som sedan bearbetas oberoende av varandra. Jag slogs av att bilden liknade 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 parallelism.

# 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 där. 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 skiftläges-känsligt och alla inbyggda identifierare är i gemener.

### Block

Paxl behandlar kod som annan data. Block kan sparar 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: *sanningsvärdens*, *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 sanningsvärd.

```
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 sanningsvärd.

```
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 används 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, där efter 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 där efter 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. Kodden 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 indenteringsstyrkt, eftersom jag märkt att det skapar svårighet och frustration för nya användare som inte är vana vid att bry sig om blanka tecken.

```
1 #!/usr/bin/env ruby
2
3 require 'lib/rdparse'
4 require 'nodes'
5
6 module Paxl
7
8   class Parser
9
10   include Paxl::Nodes
11
12   def initialize(log = false)
13
14     @parser = Rdparse::Parser.new('paxl', log) do
15
16       @scope = Hash.new
17
18       token(/\s+/)
19       token(/and|or|==|!=|<=|>=|\w+|./) { |m| m }
20
21       start :stmt_list do
22         match(:stmt, :stmt_term, :stmt_list) { |a, _, b| b = a + b }
23         match(:stmt, :stmt_term)
24         match(:stmt)
25       end
26
27       rule :stmt_term do
28         match("\n")
29         match(",")
30       end
31
32       rule :stmt do
33         match(:if_stmt)
34         match(:for_loop)
35         match(:expr)
36       end
37
38       rule :for_loop do
39         match('for', '(', :stmt_list, ')', '{', :stmt_list, '}') do
40           |_ , _, a, _, _, b, _| PForLoop.new(a, b)
41         end
42       end
43
44       rule :if_stmt do
45         match('if', '(', :stmt_list, ')', '{', :stmt_list, '}',
46             'else', '(', :stmt_list, ')') do
47             |_ , _, a, _, _, b, _, _, c, _| PIIfStatement.new(a, b, c)
48         end
49         match('if', '(', :stmt_list, ')', '{', :stmt_list, '}') do
50             |_ , _, a, _, _, b, _| PIIfStatement.new(a, b, nil)
51         end
52       end
53
54       rule :expr do
55         match('(', '|', :param_list, '|', :stmt_list, ')') do
56           |_ , _, a, _, b, _| PBlockDefinition.new(a, b)
57         end
58         match('{', :stmt_list, '}') { |_ , a, _| PBlockDefinition.new([], a) }
59         match(:identifier, '(', ')') { |a, _, _| PBlockCall.new(a, []) }
60         match(:identifier, '(', :stmt_list, ')') { |a, _, b, _| PBlockCall.new(a, b) }
61         match(:expr, '+', :term) { |a, _, b| PAddition.new(a, b) }
62       end
63     end
64   end
65
66   class Node
67     attr_accessor :value, :children
68
69     def initialize(value)
70       @value = value
71       @children = []
72     end
73
74     def add_child(child)
75       @children << child
76     end
77
78     def to_s
79       children.map { |c| c.to_s }.join(" ")
80     end
81   end
82
83   class StatementList < Node
84     def initialize
85       super()
86     end
87
88     def add_stmt(stmt)
89       add_child(stmt)
90     end
91
92     def to_s
93       children.map { |c| c.to_s }.join("\n")
94     end
95   end
96
97   class BlockDefinition < Node
98     def initialize
99       super()
100    end
101
102    def add_block(block)
103      add_child(block)
104    end
105
106    def to_s
107      children.map { |c| c.to_s }.join("\n")
108    end
109  end
110
111  class BlockCall < Node
112    def initialize
113      super()
114    end
115
116    def add_arg(arg)
117      add_child(arg)
118    end
119
120    def to_s
121      children.map { |c| c.to_s }.join(" ")
122    end
123  end
124
125  class Boolean < Node
126    def initialize(value)
127      super(value)
128    end
129
130    def to_s
131      if value == true
132        "true"
133      else
134        "false"
135      end
136    end
137  end
138
139  class Number < Node
140    def initialize(value)
141      super(value)
142    end
143
144    def to_s
145      value.to_s
146    end
147  end
148
149  class Float < Node
150    def initialize(value)
151      super(value)
152    end
153
154    def to_s
155      value.to_f.to_s
156    end
157  end
158
159  class Var < Node
160    def initialize(value)
161      super(value)
162    end
163
164    def to_s
165      value
166    end
167  end
168
169  class ParamList < Node
170    def initialize
171      super()
172    end
173
174    def add_param(param)
175      add_child(param)
176    end
177
178    def to_s
179      children.map { |c| c.to_s }.join(" ")
180    end
181  end
182
183  class Identifier < Node
184    def initialize(value)
185      super(value)
186    end
187
188    def to_s
189      value
190    end
191  end
192
193  class ComparisonOperator < Node
194    def initialize(operator)
195      super(operator)
196    end
197
198    def to_s
199      operator
200    end
201  end
202
203  class LogicalOperator < Node
204    def initialize(operator)
205      super(operator)
206    end
207
208    def to_s
209      operator
210    end
211  end
212
213  class Atom < Node
214    def initialize(value)
215      super(value)
216    end
217
218    def to_s
219      value
220    end
221  end
222
223  class Subtraction < Atom
224    def initialize(a, b)
225      super(b - a)
226    end
227
228    def to_s
229      "-"
230    end
231  end
232
233  class Multiplication < Atom
234    def initialize(a, b)
235      super(a * b)
236    end
237
238    def to_s
239      "*"
240    end
241  end
242
243  class Division < Atom
244    def initialize(a, b)
245      super(a / b)
246    end
247
248    def to_s
249      "/"
250    end
251  end
252
253  class VariableAssignment < Atom
254    def initialize(a, b)
255      super(b)
256    end
257
258    def to_s
259      "="
260    end
261  end
262
263  class VariableReference < Atom
264    def initialize(a)
265      super(a)
266    end
267
268    def to_s
269      a
270    end
271  end
272
273  class BooleanValue < Atom
274    def initialize(value)
275      super(value)
276    end
277
278    def to_s
279      value ? "true" : "false"
280    end
281  end
282
283  class Integer < Atom
284    def initialize(value)
285      super(value)
286    end
287
288    def to_s
289      value.to_s
290    end
291  end
292
293  class FloatValue < Atom
294    def initialize(value)
295      super(value)
296    end
297
298    def to_s
299      value.to_f.to_s
300    end
301  end
302
303  class String < Atom
304    def initialize(value)
305      super(value)
306    end
307
308    def to_s
309      value
310    end
311  end
312
313  class Comment < Atom
314    def initialize(value)
315      super(value)
316    end
317
318    def to_s
319      value
320    end
321  end
322
323  class Newline < Atom
324    def initialize(value)
325      super(value)
326    end
327
328    def to_s
329      "\n"
330    end
331  end
332
333  class IdentifierValue < Atom
334    def initialize(value)
335      super(value)
336    end
337
338    def to_s
339      value
340    end
341  end
342
343  class Comparison < Atom
344    def initialize(operator)
345      super(operator)
346    end
347
348    def to_s
349      operator
350    end
351  end
352
353  class Logical < Atom
354    def initialize(operator)
355      super(operator)
356    end
357
358    def to_s
359      operator
360    end
361  end
362
363  class AtomValue < Atom
364    def initialize(value)
365      super(value)
366    end
367
368    def to_s
369      value
370    end
371  end
372
373  class Block < Atom
374    def initialize(children)
375      super()
376      @children = children
377    end
378
379    def to_s
380      children.map { |c| c.to_s }.join("\n")
381    end
382  end
383
384  class BlockCallValue < Atom
385    def initialize(arguments)
386      super()
387      @arguments = arguments
388    end
389
390    def to_s
391      arguments.map { |a| a.to_s }.join(" ")
392    end
393  end
394
395  class BooleanValue < Atom
396    def initialize(value)
397      super(value)
398    end
399
400    def to_s
401      value ? "true" : "false"
402    end
403  end
404
405  class IntegerValue < Atom
406    def initialize(value)
407      super(value)
408    end
409
410    def to_s
411      value.to_s
412    end
413  end
414
415  class FloatValue < Atom
416    def initialize(value)
417      super(value)
418    end
419
420    def to_s
421      value.to_f.to_s
422    end
423  end
424
425  class StringValue < Atom
426    def initialize(value)
427      super(value)
428    end
429
430    def to_s
431      value
432    end
433  end
434
435  class CommentValue < Atom
436    def initialize(value)
437      super(value)
438    end
439
440    def to_s
441      value
442    end
443  end
444
445  class IdentifierValue < Atom
446    def initialize(value)
447      super(value)
448    end
449
450    def to_s
451      value
452    end
453  end
454
455  class ComparisonValue < Atom
456    def initialize(operator)
457      super(operator)
458    end
459
460    def to_s
461      operator
462    end
463  end
464
465  class LogicalValue < Atom
466    def initialize(operator)
467      super(operator)
468    end
469
470    def to_s
471      operator
472    end
473  end
474
475  class AtomValue < Atom
476    def initialize(value)
477      super(value)
478    end
479
480    def to_s
481      value
482    end
483  end
484
485  class BlockValue < Atom
486    def initialize(children)
487      super()
488      @children = children
489    end
490
491    def to_s
492      children.map { |c| c.to_s }.join("\n")
493    end
494  end
495
496  class BlockCallValue < Atom
497    def initialize(arguments)
498      super()
499      @arguments = arguments
500    end
501
502    def to_s
503      arguments.map { |a| a.to_s }.join(" ")
504    end
505  end
506
507  class BooleanValue < Atom
508    def initialize(value)
509      super(value)
510    end
511
512    def to_s
513      value ? "true" : "false"
514    end
515  end
516
517  class IntegerValue < Atom
518    def initialize(value)
519      super(value)
520    end
521
522    def to_s
523      value.to_s
524    end
525  end
526
527  class FloatValue < Atom
528    def initialize(value)
529      super(value)
530    end
531
532    def to_s
533      value.to_f.to_s
534    end
535  end
536
537  class StringValue < Atom
538    def initialize(value)
539      super(value)
540    end
541
542    def to_s
543      value
544    end
545  end
546
547  class CommentValue < Atom
548    def initialize(value)
549      super(value)
550    end
551
552    def to_s
553      value
554    end
555  end
556
557  class IdentifierValue < Atom
558    def initialize(value)
559      super(value)
560    end
561
562    def to_s
563      value
564    end
565  end
566
567  class ComparisonValue < Atom
568    def initialize(operator)
569      super(operator)
570    end
571
572    def to_s
573      operator
574    end
575  end
576
577  class LogicalValue < Atom
578    def initialize(operator)
579      super(operator)
580    end
581
582    def to_s
583      operator
584    end
585  end
586
587  class AtomValue < Atom
588    def initialize(value)
589      super(value)
590    end
591
592    def to_s
593      value
594    end
595  end
596
597  class BlockValue < Atom
598    def initialize(children)
599      super()
600      @children = children
601    end
602
603    def to_s
604      children.map { |c| c.to_s }.join("\n")
605    end
606  end
607
608  class BlockCallValue < Atom
609    def initialize(arguments)
610      super()
611      @arguments = arguments
612    end
613
614    def to_s
615      arguments.map { |a| a.to_s }.join(" ")
616    end
617  end
618
619  class BooleanValue < Atom
620    def initialize(value)
621      super(value)
622    end
623
624    def to_s
625      value ? "true" : "false"
626    end
627  end
628
629  class IntegerValue < Atom
630    def initialize(value)
631      super(value)
632    end
633
634    def to_s
635      value.to_s
636    end
637  end
638
639  class FloatValue < Atom
640    def initialize(value)
641      super(value)
642    end
643
644    def to_s
645      value.to_f.to_s
646    end
647  end
648
649  class StringValue < Atom
650    def initialize(value)
651      super(value)
652    end
653
654    def to_s
655      value
656    end
657  end
658
659  class CommentValue < Atom
660    def initialize(value)
661      super(value)
662    end
663
664    def to_s
665      value
666    end
667  end
668
669  class IdentifierValue < Atom
670    def initialize(value)
671      super(value)
672    end
673
674    def to_s
675      value
676    end
677  end
678
679  class ComparisonValue < Atom
680    def initialize(operator)
681      super(operator)
682    end
683
684    def to_s
685      operator
686    end
687  end
688
689  class LogicalValue < Atom
690    def initialize(operator)
691      super(operator)
692    end
693
694    def to_s
695      operator
696    end
697  end
698
699  class AtomValue < Atom
700    def initialize(value)
701      super(value)
702    end
703
704    def to_s
705      value
706    end
707  end
708
709  class BlockValue < Atom
710    def initialize(children)
711      super()
712      @children = children
713    end
714
715    def to_s
716      children.map { |c| c.to_s }.join("\n")
717    end
718  end
719
720  class BlockCallValue < Atom
721    def initialize(arguments)
722      super()
723      @arguments = arguments
724    end
725
726    def to_s
727      arguments.map { |a| a.to_s }.join(" ")
728    end
729  end
730
731  class BooleanValue < Atom
732    def initialize(value)
733      super(value)
734    end
735
736    def to_s
737      value ? "true" : "false"
738    end
739  end
740
741  class IntegerValue < Atom
742    def initialize(value)
743      super(value)
744    end
745
746    def to_s
747      value.to_s
748    end
749  end
750
751  class FloatValue < Atom
752    def initialize(value)
753      super(value)
754    end
755
756    def to_s
757      value.to_f.to_s
758    end
759  end
760
761  class StringValue < Atom
762    def initialize(value)
763      super(value)
764    end
765
766    def to_s
767      value
768    end
769  end
770
771  class CommentValue < Atom
772    def initialize(value)
773      super(value)
774    end
775
776    def to_s
777      value
778    end
779  end
780
781  class IdentifierValue < Atom
782    def initialize(value)
783      super(value)
784    end
785
786    def to_s
787      value
788    end
789  end
790
791  class ComparisonValue < Atom
792    def initialize(operator)
793      super(operator)
794    end
795
796    def to_s
797      operator
798    end
799  end
800
801  class LogicalValue < Atom
802    def initialize(operator)
803      super(operator)
804    end
805
806    def to_s
807      operator
808    end
809  end
810
811  class AtomValue < Atom
812    def initialize(value)
813      super(value)
814    end
815
816    def to_s
817      value
818    end
819  end
820
821  class BlockValue < Atom
822    def initialize(children)
823      super()
824      @children = children
825    end
826
827    def to_s
828      children.map { |c| c.to_s }.join("\n")
829    end
830  end
831
832  class BlockCallValue < Atom
833    def initialize(arguments)
834      super()
835      @arguments = arguments
836    end
837
838    def to_s
839      arguments.map { |a| a.to_s }.join(" ")
840    end
841  end
842
843  class BooleanValue < Atom
844    def initialize(value)
845      super(value)
846    end
847
848    def to_s
849      value ? "true" : "false"
850    end
851  end
852
853  class IntegerValue < Atom
854    def initialize(value)
855      super(value)
856    end
857
858    def to_s
859      value.to_s
860    end
861  end
862
863  class FloatValue < Atom
864    def initialize(value)
865      super(value)
866    end
867
868    def to_s
869      value.to_f.to_s
870    end
871  end
872
873  class StringValue < Atom
874    def initialize(value)
875      super(value)
876    end
877
878    def to_s
879      value
880    end
881  end
882
883  class CommentValue < Atom
884    def initialize(value)
885      super(value)
886    end
887
888    def to_s
889      value
890    end
891  end
892
893  class IdentifierValue < Atom
894    def initialize(value)
895      super(value)
896    end
897
898    def to_s
899      value
900    end
901  end
902
903  class ComparisonValue < Atom
904    def initialize(operator)
905      super(operator)
906    end
907
908    def to_s
909      operator
910    end
911  end
912
913  class LogicalValue < Atom
914    def initialize(operator)
915      super(operator)
916    end
917
918    def to_s
919      operator
920    end
921  end
922
923  class AtomValue < Atom
924    def initialize(value)
925      super(value)
926    end
927
928    def to_s
929      value
930    end
931  end
932
933  class BlockValue < Atom
934    def initialize(children)
935      super()
936      @children = children
937    end
938
939    def to_s
940      children.map { |c| c.to_s }.join("\n")
941    end
942  end
943
944  class BlockCallValue < Atom
945    def initialize(arguments)
946      super()
947      @arguments = arguments
948    end
949
950    def to_s
951      arguments.map { |a| a.to_s }.join(" ")
952    end
953  end
954
955  class BooleanValue < Atom
956    def initialize(value)
957      super(value)
958    end
959
960    def to_s
961      value ? "true" : "false"
962    end
963  end
964
965  class IntegerValue < Atom
966    def initialize(value)
967      super(value)
968    end
969
970    def to_s
971      value.to_s
972    end
973  end
974
975  class FloatValue < Atom
976    def initialize(value)
977      super(value)
978    end
979
980    def to_s
981      value.to_f.to_s
982    end
983  end
984
985  class StringValue < Atom
986    def initialize(value)
987      super(value)
988    end
989
990    def to_s
991      value
992    end
993  end
994
995  class CommentValue < Atom
996    def initialize(value)
997      super(value)
998    end
999
1000   def to_s
1001     value
1002   end
1003 
```

```
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| PIInteger.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
```

```
1 #!/usr/bin/env ruby
2 module Paxl
3   module Nodes
4
5     class Node
6     end
7
8     class PStatementList < Node
9
10    def initialize(statement)
11      if statement.kind_of? Array
12        @statement_list = statement
13      else
14        @statement_list = [statement]
15      end
16    end
17
18    def +(statement_list)
19      self.class.new @statement_list + statement_list.list
20    end
21
22    def list
23      @statement_list
24    end
25
26    def eval(scope)
27      result = nil
28      @statement_list.each do |statement|
29        result = statement.eval(scope)
30      end
31      return result
32    end
33
34  end
35
36  class PForLoop < Node
37
38    def initialize(control, statements)
39      @control, @statements = control, statements
40    end
41
42    def eval(scope)
43      my_scope = Paxl::Scope.new(scope)
44      init_stmt, test_stmt, iter_stmt = @control.list
45      init_stmt.eval(my_scope)
46      return_value = nil
47      while test_stmt.eval(my_scope) do
48        return_value = @statements.eval(my_scope)
49        iter_stmt.eval(my_scope)
50      end
51      return_value
52    end
53
54  end
55
56  class PIfStatement < Node
57
58    def initialize(test, if_true, if_false)
59      @test, @if_true, @if_false = test, if_true, if_false
60    end
61
```

```
62    def eval(scope)
63      test_result = @test.eval(scope)
64      if test_result
65        @if_true.eval(scope)
66      else
67        @if_false
68        @if_false.eval(scope)
69      else
70        nil
71      end
72    end
73  end
74
75  class PBlockDefinition < Node
76
77    attr_reader :parameters, :statement_list, :block_scope
78
79    def initialize(parameters, statements)
80      @parameters, @statement_list = parameters, PStatementList.new(statements)
81    end
82
83    def eval(scope)
84      @block_scope = scope.clone
85      @block_scope["this"] = self
86      self
87    end
88
89  end
90
91  class PBlockCall < Node
92
93    def initialize(identifier, arguments)
94      @identifier, @arguments = identifier, arguments
95    end
96
97    def eval(scope)
98      block_def = scope[@identifier]
99      if not block_def.kind_of? Paxl::Nodes::PBlockDefinition
100        return "Error! Attempting to use a non-block as a block."
101      end
102      my_scope = block_def.block_scope.clone
103      if @arguments.kind_of? Paxl::Nodes::PStatementList
104        arguments = Array.new(@arguments.list)
105      end
106      block_def.parameters.each do |param|
107        my_scope[param] = arguments.pop().eval(scope)
108      end
109      block_def.statement_list.eval(my_scope)
110    end
111
112  end
113
114  class PVariableAssignment < Node
115
116    def initialize(identifier, value)
117      @identifier, @value = identifier, value
118    end
119
120    def eval(scope)
121
```

```
1 require 'test/unit'
2 require 'paxl'
3
4 class TestPaxl < Test::Unit::TestCase
5
6 # all tests run using the same global scope
7 def initialize(arg)
8   super(arg)
9   @paxl = Paxl::Parser.new
10  @paxl_scope = Paxl::Scope.new(nil)
11 end
12
13 def assert_returns(value, code)
14   assert_equal(value, @paxl.parse(code).eval(@paxl_scope))
15 end
16
17 def test_y_combinator
18   assert_returns 120, "
19     y = { |g|
20       f = { |x|
21         { |arg|
22           gx = g(x(x));
23           gx(arg)
24         }
25       };
26       f(g)
27     };
28     fg = { |cb|
29       { |arg|
30         if (arg == 0) { 1 }
31         else { arg * (cb(arg - 1)) }
32       }
33     };
34     factorial = y(fg);
35     factorial(5)"
36   end
37
38 def test_this_keyword
39   assert_returns 120, "
40     factorial = { |x|
41       if (x == 0) { 1 }
42       else { x * (this(x - 1)) }
43     };
44     factorial(5)"
45   end
46
47 def test_for_loops
48   assert_returns 45, "a = 0; for (i = 0; i < 10; i = i + 1) { a = a + i }; a"
49   end
50
51 def test_if_statements
52   assert_returns 10, "a = 5; if (a < 10) { 10 } else { a }"
53   assert_returns 15, "a = 15; if (a < 10) { 10 } else { a }"
54   assert_returns 10, "a = 10; if (a == 10) { 10 }"
55   end
56
57 def test_blocks
58   assert_returns 200, "a = { |b, c| b * c }; a(10; 20)"
59   assert_returns 250, "n = 50; a = { |b, c| b * c + n }; a(10; 20)"
60   assert_returns 10, "a = { 10 }; a();"
61   assert_returns 10, "a = { |x, y, z| (x * y) - z }; a(5; 4; 10)"
```

```
62   end
63
64 def test_variable_assignment
65   assert_returns 10, "a = 10"
66   assert_returns 10, "a = 10"
67   assert_returns 5, "b = 5"
68   assert_returns 10, "a"
69   assert_returns 5, "b"
70   assert_returns 50, "a * b"
71 end
72
73 def test_basic_math
74   assert_returns 10, "5 + 5"
75   assert_returns 10, "15 - 5"
76   assert_returns 10, "100 / 10"
77   assert_returns 10, "5 * 2"
78 end
79
80 def test_logical_expressions
81   assert_returns true, "true"
82   assert_returns false, "false"
83   assert_returns true, "true and true"
84   assert_returns false, "true and false"
85   assert_returns true, "false or true"
86   assert_returns true, "not false"
87 end
88
89 def test_comparisons
90   assert_returns true, "10 == 10"
91   assert_returns false, "1 == 2"
92   assert_returns true, "1 != 2"
93   assert_returns true, "1 < 2"
94   assert_returns true, "1 <= 2"
95   assert_returns false, "1 >= 2"
96   assert_returns false, "1 > 2"
97 end
98
99 def test_nested_expressions
100  assert_returns true, "(true or false) and (true and true)"
101  assert_returns true, "(10 < 20) and (20 >= 10)"
102  assert_returns true, "a = ( (true or false) and (true and true) )"
103
104
105 end
```

```
1#!/usr/bin/env ruby
2require 'rubygems'
3require 'spreadsheet'
4require 'paxl'
5
6filename = ARGV.shift
7if not filename
8  abort "I expect a .xls filename as an argument"
9end
10book = Spreadsheet.open filename
11sheet = book.worksheet 0
12number_of_threads = 4
13threads = []
14
15sheet.each_slice(number_of_threads) do |rows|
16  threads << Thread.new(rows) do |rows|
17    p = Paxl::Parser.new
18    rows.each do |row|
19      row_scope = Paxl::Scope.new(nil)
20      column_labels = ('A'..'Z').to_a
21      values = []
22      row.each do |cell|
23        code = "#{cell}"
24        lbl = column_labels.shift
25        result = p.parse(code).eval(row_scope)
26        row_scope[lbl] = result
27        values << result
28      end
29      puts "#{row.idx},#{values.join(',')}"
30    end
31  end
32end
33
34threads.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 PLiteral < Node
141
142   def initialize(value)
143     @value = value
144   end
145
146   def eval(scope)
147     @value
148   end
149
150   def to_s
151     @value.to_s
152   end
153
154   def to_i
155     @value.to_i
156   end
157
158 end
159
160 class PLogicalExpression < 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 "and"
170       return PBoolean.new( (a and b) ).eval(scope)
171     when "or"
172       return PBoolean.new( (a or b) ).eval(scope)
173     when "not"
174       return PBoolean.new( (not a) ).eval(scope)
175     end
176   end
177
178 end
179
180 class PComparison < Node
181
182   def initialize(a, op, b)
183     @a, @op, @b = a, op, b
184   end
185
186   def eval(scope)
187     a, b = @a.eval(scope), @b.eval(scope)
188     case @op
189     when "==""
190       return PBoolean.new( (a == b) ).eval(scope)
191     when "!="
192       return PBoolean.new( (a != b) ).eval(scope)
193     when "<="
194       return PBoolean.new( (a <= b) ).eval(scope)
195     when ">="
196       return PBoolean.new( (a >= b) ).eval(scope)
197     when "<"
198       return PBoolean.new( (a < b) ).eval(scope)
199     when ">"
200       return PBoolean.new( (a > b) ).eval(scope)
201     end
202   end
203
204 end
```

```
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  class PFloat < Node
246
247    def initialize(value)
248      @value = value.to_f
249    end
250
251    def eval(scope)
252      @value
253    end
254
255  end
256
257  class PBoolean < Node
258
259    def initialize(value)
260      if value == true or value == false
261        @value = value
262      else
263        @value = (value != 0) # false if value == 0, true otherwise
264      end
265    end
266
267    def eval(scope)
268      @value
269    end
270
271  end
272
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
```