shl

shorthand language

Fredrik Jonsén frejo105
Daniel Eriksson daner045

TDP019 - Projekt: Dataspråk
Linköpings universitet

Sammanfattning

Detta projekt har som mål att konstruera och implementera ett eget programmeringsspråk för kursen TDP019 under vårterminen 2015. Tanken bakom vårt språk, som vi har namngett shorthand language, är, precis som man kan gissa av namnet, att ha en så kort syntax som möjligt. Sådant som nyckelord har därmed i de flesta fall bytts ut med en enda symbol, och så många delar som möjligt i statements är frivilliga.

Språket är interpreterat och all kod som parsern bygger på är skriven i Ruby.

Innehållsförteckning

1 Inledning

Detta projekt är utfört av Fredrik Jonsén och Daniel Eriksson, IP-programmet årskurs 1, under vårterminen 2015. Målet med projektet är att utveckla ett programmeringsspråk. Det språk vi har utvecklat är ett interpreterat språk, där syntaxen är tänkt att vara så kort som möjligt, och därmed få ett språk som är “effektivt” att skriva. Inspiration är taget huvudsakligen från andra interpreterade språk, såsom Python och Ruby.

Detta dokument är tänkt att fungera som användarhandledning med exempel, teknisk genomgång av språket och parsningens struktur, samt reflektion för projektet och tankarna bakom språkets syntax och evaluering vid körning.

2 Användarhandledning

I det här avsnittet finns allt man behöver för att komma igång att skriva kod i shl. Förkunskaper i programmering behövs.

2.1 Användning

Ruby version 2 eller senare krävs för att kunna köra shl. Språket körs genom att köra parser-filen i Ruby med en textfil för språket att läsa in som argument.

Exempel:

```
> ruby shlParse.rb fil.shl
```

2.2 Datatyper

shl har 6 olika datatyper:

- Boolean
- Integer
- Float
- String
- Array
- Hash

2.3 Aritmetiska uttryck

Operatorer:

+	addition
-	subtraktion
*	multiplikation
/	division
**	exponentiering
//	heltalsdivision

Exempel:

```
1 + 1;   # 2
2 ** 4;  # 16
10 // 4; # 2
```

2.4 Jämförelse- och logikuttryck

Operatorer för jämförelse:

==	lika med
!=	skilt från
<	mindre än
>	större än
<=	mindre än eller lika med
>=	större än eller lika med

Operatorer för logik:

&&	och
||	eller

Exempel:

```
1 < 4;    # true
4 <= 3; # false
3 <= 3; # true
(1 < 2) && (1 == 2); # false
(1 < 2) || (1 == 2);    # true
```

2.5 Tilldelning

Tilldelning i shl är dynamiskt typat så vilken datatyp som helst kan tilldelas till en variabel utan att specificera datatypen.

Exempel:

```
bool = true;
i_num = 3;
f_num = 3.5;
str = "three";
nums = [1,2,3];
i_to_s = {1:"one", 2:"two", 3:"three"};
```

shl har symboler för att tilldela ett "standardvärde" av en typ för att lätt kunna visa att man skapar en variabel innan användning.

Exempel:

```
b :b;  # Boolean med värdet false tilldelas.
i :i;  # Integer med värdet 0 tilldelas.
f :f;  # Float med värdet 0.0 tilldelas.
s :s;  # Tom sträng ( "" ) tilldelas.
a :a;  # Tom array ( [] ) tilldelas.
h :h;  # Tom hash ( {} ) tilldelas.
```

shl har två operatorer för att öka en integer eller float med 1. Dessa kan skrivas före eller efter en variabel:

```
++a  # a ökar med 1 innan användning.
a++  # a ökar med 1 efter användning.
```

Exempel:

```
a = 5;
a++; # 6
a--;  # 5
```

2.6 Funktioner

Funktioner skrivs med ett @-tecken följt av funktionsnamn, sedan parenteser som innehåller funktionens parametrar, om några behövs. En parameter kan tilldelas ett värde och då behöver man inte ger den något värde vid funktionsanropet. Detta kallas för en default-parameter och de måste hållas till höger bland alla parametrar.

De satser som funktionen utför skrivs inom ett kodblock som defineras av symbolerna { och }, för att returnera ett värde från funktionen används <- innan ett uttryck. Funktionen avbryts när den returnerar.

Exempel:

```
@power(a, b=2) { <- a ** 2; }

power(2);      # 2 ** 2
power(2, 3);  # 2 ** 3
```

2.7 Villkorssatser

if-satser består av villkor och kodblock som körs om villkoret uppfylls.

~i *if*, början av en if-sats, det första villkoret.
~ei *else if*, om den övre villkoret inte uppfylldes, så testas detta.
Det kan finnas hur många *else if*s som helst.
~e *else*, om inget av de övre villkoren uppfylldes så körs detta kodblock.
Kan bara finnas en.

Exempel:

```
~i a < b { pl("a is less than b."); }
~ei a > b { pl("a is greater than b."); }
~e { pl("a and b are equal."); }
```

2.8 Loopar

Det finns två olika typer av loopar i shl:

~w while-loop, som innehåller ett villkor och ett kodblock som körs så länge villkoret uppfylls.

Exempel:

```
a :i;
~w a < 10 { pl(a); ^a++; } #skriver ut 0 till 9
```

^ operatorn används framför en variabel för att leta efter den variabeln från och med räckvidden (scope) över den räckvidd man koden befinner sig i.

~f for-loop, som innehåller en tilldelning, ett villkor för loopen såväl som ett uttryck för att ändra på den tilldelade variabeln för varje loop som sker.

Exempel:

```
~f a :i; a < 10; a++; { pl(a); }
```

2.9 Elementoperator

Med elementoperatorn kan man:

- Nå ett element i en array:

```
a = [1,2,3];
a[2];       # returnerar 3.
a[0] = 5; # a är nu [5,2,3].
```

- Nå värdet av ett nyckelvärde par i en hash:

```
h = {1:"one", 2:"two", 3:"three"};
h[1];              # returnerar "one".
h[4] = "four";  # nytt nyckel-värde par.
```

- Nå ett tecken i en sträng:

```
s = "abcde";
s[4];        # returnerar "e".
s[0] = "q" # s har värdet "qbcde".
```

2.10 Klasser

En klassdefinition är väldigt lik en funktionsdefinition, en klassdefinition ser precis ut som en funktions definition fast med ett !-tecken istället. Alla funktioner och variabler deklarerade inom kodblocket blir medlemmar till klassen. När en klass tilldelas till en variabel körs all kod i kodblocket, alltså kan man skriva kod som ska köras vid skapandet direkt i blocket, som en konstruktor.

Medlemmar nås genom att använda punktoperatorn (.).

Exempel:

```
!Rectangle(x,y,w,h)
{
	@area() { <- w * h; }
}
r = Rectangle(1,1,3,4);
r.area();  # returnerar 12
r.w = 4;
r.area();  # returnerar 16
```

2.11 Inbyggda funktioner

Utskrift utan ny rad efter:
```
p(x);
```

Utskrift med ny rad efter:
```
pl(x);
```

Inbyggda medlemsfunktioner:

Upprepa en string x gånger:
```
str.times(3);
```

Kolla om ett element finns i en array:
```
arr.includes(4);
```

Kolla om en nyckel finns i en hash:
```
hash.has_key(”key”);
```

3 Systemdokumentation

Det här avsnittet beskriver hur språket är uppbyggt och vad det består av.

3.1 Översikt

shl använder den givna klassen rdparse, en *recursive descent parsar* för parsning av koden.Ett abstrakt syntaxträd skapas då, bestående av noder. Varje typ av *statement* (allt ifrån själva programmet till en ensam konstant) är en nod. Efter detta evalueras sedan syntaxträdet från början av programmet.

Hela systemet är uppdelat i fem filer.
rdparse.rb - detta är den givna parsern, som med hjälp av regler parsar koden.
shlParse.rb - i denna fil anges alla de regler som rdparse använder.
nodes.rb - i denna fil finns alla noder som bygger upp det abstrakta syntaxträdet, samt Scope-klassen.
helpers.rb - olika funktioner för att underlätta skapandet av det abstrakta syntaxträdet.
builtins.rb - olika funktioner som kan anropas av ett shl-program.

3.2 Grammatik

```
<begin> ::= <stmt_list>

<stmt_list> ::= <stmt> [<stmt_list>]

<stmt> ::= <expr> ';'
        | <if_stmt>
        | <for_stmt>
        | <while_stmt>
        | <class_def>
        | <function_def>
        | <return> ';'
        | '!->' ';'      # continue
        | '->!' ';'      # break

<expr> ::= <assignment>
        | <conversion>
        | <bool_expr>
        | <comparison>
        | <expr_call>
        | <arith_expr>
        | <identifier>
        | type
        | '!' <expr>
        | '(' <expr> ')'
```

```
<unary_expr> ::= <unary_op> ['^'] <identifier>
               | ['^'] <identifier> <unary_op>

<unary_op> ::= '++' | '--'

<conversion> ::= <identifier> '->' <type_dec>

<bool_expr> ::= <bool_expr> '&&' <expr>
               | <bool_expr> '||' <expr>
               | <bool>

<expr_call> ::= <identifier> '(' [<arg_list>] ')'

<if_stmt> ::= '~i' <expr> <cond_body> [<elseif_list>]['~e' <cond_body>]

<elseif_list> ::= [<elseif_list>] <elseif>

<elseif> ::= '~ei' <expr> <cond_body>

<for_stmt> ::= '~f' <assignment> ';' <expr> ';' <expr> <cond_body>
       | '~f' <assignment> ';' <expr> <cond_body>
       | '~f' <expr> ';' <expr> <cond_body>
       | '~f' <expr> ';' ';' <expr> <cond_body>
       | '~f' <expr> <cond_body>
       | '~f' <cond_body>

<while_stmt> ::= '~w' [<expr>] <cond_body>

<cond_body> ::= '{' <stmt_list '}'
               | <stmt>

<arg_list> ::= <expr> [',' <arg_list>]

<param_list> ::= <identifier> ',' [<param_def_list> | <param_list>]
               | <param_def_list>
               | <identifier>

<param_def_list> ::= <identifier> '=' <expr> [',' <param_def_list>]

<class_def> ::= '!' <identifier> ['(' [<param_list>] ')'] '{' <stmt_list> '}'

<function_def> ::= '@' <identifier> ['(' [<param_list>] ')'] '{' <stmt_list> '}'
```

```
<value> ::= <type> | <identifier>

<identifier> ::= <identifier> '.' <identifier>
               | <type> '.' <identifier>
               | <identifier> '[' <value> ']'
               | <name>

<name> ::= /[\wÅÄÖåäö][\w\d_åäöÅÄÖ]*/

<comp_op> ::= '==' | '<=' | '>=' | '!=' | '<' | '>'

<comparison> ::= <arith_expr> <comp_op> <arith_expr>

<type_dec> ::= /:[ifsahb]/

<arith_op> ::= '+' | '-'

<arith_expr> ::= <arith_expr> <arith_op> <term>
               | <term> <arith_op> <term>
               | <term>

<term_op> ::= '//' | '*' | '/'

<term> ::= <term> <term_op> <pow>
       | <pow> <term_op> <term>
       | <pow>

<pow> ::= <pow> '**' <factor>

<factor> ::= '(' <arith_expr> ')'
       | '-' <term>
       | <unary_expr>
       | <type>
       | <expr_call>
       | <identifier>

<assignment> ::= <expr_assignment> | <type_assignment>

<expr_assignment> ::= ['^']<identifier> '=' <expr>

<type_assignment> ::= <identifier> <type_dec>
```

<return> ::= '<-' [<expr>]

<type> ::= <bool> | <int> | <float> | <string> | <array> | <hash> | <nil>

<hash_arg_list> ::= <hash_arg> [',' <hash_arg_list>]

<hash_arg> ::= <value> ':' <value>

<hash> ::= '{' [<hash_arg_list>] '}'

<array> ::= '[' <arg_list> ']'

<hash> ::= '{' [<hash_arg_list>] '}'

<array> ::= '[' [<arg_list>] ']'

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

<float> ::= Float

<int> ::= Integer

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

<nil> ::= 'nil'

3.3 Lexikalisk analys

Den lexikaliska analysen är första steget. Här bildas *tokens* av de enskilda delarna i programmet. Tokens är operatorer, numeriska tal, strängar, nyckelord och dylikt. I ordning hanteras:

1. Whitespace, vilket ignoreras
2. Kommentarer, vilka ignoreras
3. Flyttal
4. Heltal
5. Strängar
6. Identifiers och nyckelord
7. Typtilldelningar
8. Nyckelord för loopar och conditionals
9. Operatorer som består av fler än ett tecken
10. Allt annat

3.4 Parsning

I detta steg används de tokens som skapades i den lexikaliska analysen, tillsammans med grammatiken ovan, för att bygga ett *abstrakt syntaxträd*. Här behandlas även vissa statements direkt, så som att översätta en typdeklaration till ett standardvärde. Exempelvis, då en variabler deklareras till en integer, ges variabeln värdet 0.

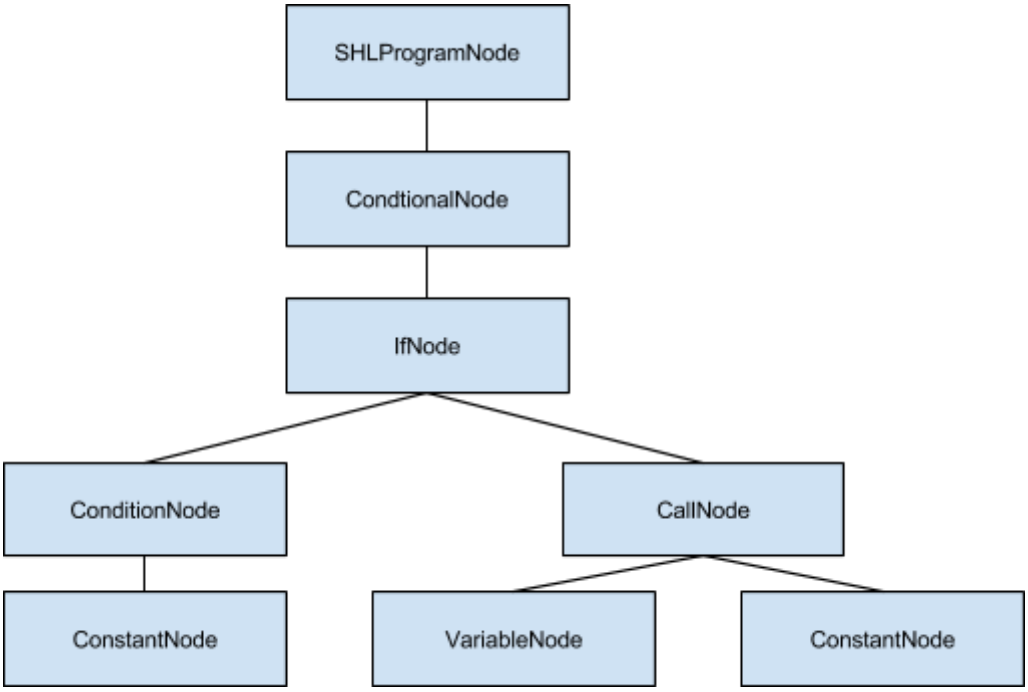
3.5 Abstrakt Syntaxträd

Det abstrakta syntaxträdet består av en mängd nästlade noder. Överst finns en SHLProgramNode, som har en lista på alla andra noder. Alltså består allt ifrån själva programmet, till mer övergripande statements, till de fundamentala delarna så som konstanter, av noder.

När hela programmet har parsats evalueras sedan det abstrakta syntaxträdet, genom att helt enkelt köra den gemensamma funktioner evaluate som alla noder har. I början går SHLProgramNode igenom sin lista med statements, och kallar evaluate på alla. I vissa fall, så som *conditionals*, kontrolleras först ett villkor för att se om noden ska evalueras.

Exempel:

```
~i true
{
  pl(2);
}
```



3.6 Testning

Hela system testas med ett antal automatiska enhetstester, som finns i mappen test_files. Testerna kan antingen köras i sin helhet genom filen run_all_tests.rb, eller individuellt genom att köra någon av testfilerna direkt. När testerna körs bör en text visas om något test av någon anledning inte skulle gå igenom.

4 Erfarenheter och reflektion

Det kanske första vi insåg när vi började med språket var just hur svårt det är att komma på något särskilt unikt; något vi inte sett i andra språk. Mycket som till en början känns som en bra och unik idé visar sig snabbt istället vara mycket opraktiskt. Ett exempel är att vi till en början hade tänkt ha signifikant whitespace, något vi sedan ångrade oss om när vi insåg hur svårt det i vissa fall kunde vara att korrekt parsa ett statement där flera expressions skulle skrivas på samma rad.

Vårt mål med projektet blev att skapa ett språk med så kort syntax som möjligt. Huvudsakligen betydde detta att byta ut nyckelord mot symboler, och möjligheten att skriva statements så kort som möjligt; exempelvis att kunna skippa delar i ett for-statement som inte behövdes. Till viss del har vi lyckats med detta, även om det, igen, i vissa fall visade sig vara mycket opraktiskt. I så många fall som möjligt har vi valt att föredra få tecken över läsbarhet, eftersom detta var vårt mål. Med rätt tecken så tycker vi att läsbarheten för det mesta behålls, det blev inte så svårläst som vi trodde.

Ett av de kanske mest återkommande problemen, som tog en hel del tid i början av projektet, var att vi hade svårt att förstå exakt hur *rdparse* fungerade. Eftersom det inte fanns någon dokumentation fick vi många gånger helt enkelt försöka experimentera oss fram till hur det fungerade, vilket slösade bort många timmar och skapade en hel del frustration.

Vi känner trots allt att det gick relativt bra att skriva grammatiken till vårt språk. Även om det i början var svårt att se till att alla delar kom med, så var det sällan några problem att helt enkelt successivt lägga till nya idéer och funktioner som vi kände skulle passa.

Desto mer knepigare var kanske implementationen av det abstrakta syntaxträdet. En hel del tid gick åt till att försöka komma fram till hur implementationen faktiskt skulle fungera, och det var alltså mycket tid då vi inte fick något direkt praktiskt, alltså någon kod, gjord.

På grund av att vi jobbade väldigt mycket separat så tog det ofta extra tid att förstå den andra personens ändringar innan man fortsatte själv. Tack vare versionshantering så kunde vi lätt se vad som hade ändrats, men vissa saker kunde ha gått smidigare om vi hade spenderat mer tid med parprogrammering.

5 Bilagor

Bilagor med projektkod, rdparse.rb och alla testfall inkluderas inte.

5.1 shlParse.rb

```
#! /usr/bin/ruby

# coding: utf-8
require_relative './rdparse'
require_relative './nodes'
require_relative './helpers'


class SHLParse
 def initialize
   @shlp = Parser.new('shorthand language') do


    # LEXER
    token(/\s+/)
    token(/#>[^<]*<#/)       # start-stop comment
    token(/#.*$/)            # single line comment
    token(/\d+\.\d+/) { |m| m.to_f } # float
    token(/\d+/)     { |m| m.to_i }    # int
    token(/"[^"]*"/) { |m| m }         # strings
    token(/[\wÅÄÖåäö][\w\d_åäöÅÄÖ]*/) { |m| m } # identifiers
    token(/:[ifsahb]/) { |m| m }        # type assignments
    token(/~ei|~[iewf]/) { |m| m }    # if / loops
    token(/\!->|->\!/) { |m| m }       # interrupt keywords
    token(/==|<=|>=|!=|\*\*|\/\/|<-|->|\+\+|--|&&|\|\|/) { |m| m }
    token(/./) { |m| m }               # symbol


    # PARSER
    start :begin do
     match(:stmt_list) { |sl| SHLProgramNode.new(sl) }
    end


    rule :stmt_list do
     match(:stmt, :stmt_list) { |s, sl| [s].concat(sl) }
     match(:stmt) { |s| [s] }
    end


    rule :stmt do
     match(:expr, ';') { |s, _| s }
     match(:if_stmt)
```

```
  match(:for_stmt)
  match(:while_stmt)
  match(:class_def)
  match(:function_def)
  match(:return, ';')
  match('!->', ';') { InterruptNode.new(:continue) } # continue
  match('->!', ';') { InterruptNode.new(:break) }      # break
end

rule :expr do
  match(:assignment)
  match(:conversion)
  match(:bool_expr)
  match(:comparison)
  match(:expr_call)
  match(:arith_expr)
  match(:identifier)
  match(:type)
  match('!', :expr) { |_, b| NegationNode.new(b) }
  match('(', :expr, ')')
end

rule :unary_expr do
  match(:unary_op, :identifier) do |op, expr|
    UnaryExprNode.new(expr, op, false)
  end
  match(:unary_op, '^', :identifier) do |op, _, expr|
    UnaryExprNode.new(expr, op, false, true)
  end
  match(:identifier, :unary_op) do |expr, op|
    UnaryExprNode.new(expr, op, true)
  end
  match('^', :identifier, :unary_op) do |_, expr, op|
    UnaryExprNode.new(expr, op, true, true)
  end
end

rule :conversion do
  match(:identifier, '->', :type_dec) do |i, _, t|
    ConversionNode.new(i, t)
  end
end
```

```
rule :bool_expr do
 match(:bool_expr, '&&', :expr) { |a, _, b| BoolExprNode.new(a, b, '&&') }
 match(:bool_expr, '||', :expr) { |a, _, b| BoolExprNode.new(a, b, '||') }
 match(:bool)
end

rule :expr_call do
 match(:identifier, '(', :arg_list, ')') { |i, _, al| CallNode.new(i, al)}
 match(:identifier, '(', ')') { |i| CallNode.new(i, []) }
end

rule :if_stmt do
 match('~i', :expr, :cond_body, :elseif_list,  '~e', :cond_body) \
 do |_, i_cond, i_body, elseifs, _, e_body|
  if_statement_handler(i_body, i_cond, elseifs, e_body)
 end

 match('~i', :expr, :cond_body, :elseif_list) \
 do |_, i_cond, i_body, elseifs|
  if_statement_handler(i_body, i_cond, elseifs)
 end

 match('~i', :expr, :cond_body, '~e', :cond_body) \
 do |_, i_cond, i_body, _, e_body|
  if_statement_handler(i_body, i_cond, [], e_body)
 end

 match('~i', :expr, :cond_body) { |_, c, b| if_statement_handler(b, c) }
end

rule :elseif_list do
 match(:elseif_list, :elseif) { |list, ei| [ei].concat(list) }
 match(:elseif) { |a| [a] }
end

rule :elseif do
 match('~ei', :expr, :cond_body) do |_, c, b|
  IfNode.new(BlockNode.new(b), c)
 end
end
```

```
rule :for_stmt do
  # No version with only assignment exists, as it is impossible
  # to tell whether an assignment is meant to be just an assigment
  # or the condition

  # All specified
  match('~f', :assignment, ';', :expr, ';', :expr, :cond_body) \
  do |_, a, _, c, _, i, body|
    for_statement_handler(body, assignment: a, cond: c, inc: i)
  end
  # Assignment and Condition specified
  match('~f', :assignment, ';', :expr, :cond_body) do
    |_, a, _, c, body|
    for_statement_handler(body, assignment: a, cond: c)
  end
  # Condition and Incement specified
  match('~f', :expr, ';', :expr, :cond_body) do |_, c, _, i, b|
    for_statement_handler(b, cond: c, inc: i)
  end
  # Assignment and Incement specified
  match('~f', :assignment, ';', ';', :expr, :cond_body) do
    |_, a, _, _, i, body|
    for_statement_handler(body, assignment: a, inc: i)
  end
  # Condition specified
  match('~f', :expr, :cond_body) do |_, c, body|
    for_statement_handler(body, cond: c)
  end
  # None specified
  match('~f', :cond_body) do |_, body|
    for_statement_handler(body)
  end
end

rule :while_stmt do
  match('~w', :expr, :cond_body) do |_, e, c|
    WhileNode.new(e, BlockNode.new(c))
  end
  match('~w', :cond_body) do |_, c|
    WhileNode.new(ConstantNode.new(true), BlockNode.new(c))
  end
end
```

```
rule :cond_body do
 match('{', :stmt_list, '}') { |_, s, _| s }
 match(:stmt) { |x| [x] }
end

rule :arg_list do
 match(:expr, ',', :arg_list) { |e, _, al| [e].concat(al) }
 match(:expr) { |e| [e] }
end

rule :param_list do
 match(:identifier, ',', :param_def_list) do |i, _, pdl|
  [[i.name, :nv]].concat(pdl)
 end
 match(:identifier, ',', :param_list) do |i, _, pl|
  [[i.name, :nv]].concat(pl)
 end
 match(:param_def_list)
 match(:identifier) { |i| [[i.name, :nv]] }
end

rule :param_def_list do
 match(:identifier, '=', :expr, ',', :param_def_list) { |i,_,e,_,pdl| [[i.name,e]].concat(pdl) }
 match(:identifier, '=', :expr) { |i, _, e| [[i.name,e]] }
end

rule :class_def do
 match('!', :identifier,'(', :param_list, ')', '{', :stmt_list, '}') do |_,i,_,pl,_,_,sl|
  CallableDefNode.new(i.name, :class, pl, BlockNode.new(sl))
 end
 match('!', :identifier, '{', :stmt_list, '}') do |_,i,_,sl|
  CallableDefNode.new(i.name, :class, [], BlockNode.new(sl))
 end
 match('!', :identifier, '(', ')', '{', :stmt_list, '}') do |_, i, _, _, _, sl|
  CallableDefNode.new(i.name, :class, [], BlockNode.new(sl))
 end
end

rule :function_def do
 match('@', :identifier, '(', :param_list, ')', '{', :stmt_list, '}') do |_,i,_,pl,_,_,sl|
  CallableDefNode.new(i.name, :func, pl, BlockNode.new(sl))
 end
```

```
  match('@', :identifier, '(', ')', '{', :stmt_list, '}') do |_,i,_,_,_,sl|
    CallableDefNode.new(i.name, :func, [], BlockNode.new(sl))
  end
  match('@', :identifier, '{', :stmt_list, '}') do |_, i, _, sl, _|
    CallableDefNode.new(i.name, :func, [],  BlockNode.new(sl))
  end
end

rule :value do
  match(:type)
  match(:identifier)
end

rule :identifier do
  match(:identifier, '.', :identifier) { |i1, _, i2| MemberNode.new(i1, i2) }
  match(:type, '.', :identifier) { |i1, _, i2| MemberNode.new(i1, i2) }
  match(:identifier, '[', :value, ']') { |n, _, t, _| BracketCallNode.new(n, t) }
  match(:name) { |n| VariableNode.new(n) }
end

rule :name do
  match(/[\wÅÄÖåäö][\w\d_åäöÅÄÖ]*/) { |a| a }
end

rule :unary_op do
  match('++')
  match(/--/)
end

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

rule :comparison do
  match(:arith_expr, :comp_op, :arith_expr) do |a, op, b|
    ComparisonNode.new(a, b, op)
  end
```

```
end

rule :type_dec do
 match(':i') { ConstantNode.new(0) }
 match(':f') { ConstantNode.new(0.0) }
 match(':s') { ConstantNode.new('') }
 match(':a') { ConstantNode.new([]) }
 match(':h') { ConstantNode.new({}) }
 match(':b') { ConstantNode.new(false) }
end

rule :arith_op do
 match('+')
 match('-')
end

rule :arith_expr do
 match(:arith_expr, :arith_op, :term) do |a, op, b|
  ArithmeticNode.new(a, b, op)
 end
 match(:term, :arith_op, :term) do |a, op, b|
  ArithmeticNode.new(a, b, op)
 end
 match(:term)
end

rule :term_op do
 match('//')
 match('*')
 match('/')
end

rule :term do
 match(:term, :term_op, :pow) do |a, op, b|
  ArithmeticNode.new(a, b, op)
 end
 match(:pow, :term_op, :term) do |a, op, b|
  ArithmeticNode.new(a, b, op)
 end
 match(:pow)

end
```

```
rule :pow do
 match(:pow, '**', :factor) { |a, _, b| ArithmeticNode.new(a, b, '**') }
 match(:factor)
end

rule :factor do
 match('(', :arith_expr, ')') { |_, b, _| b }
 match('-', :term) { |_, v| UnaryExprNode.new(v, '-', false) }
 match(:unary_expr)
 match(:type)
 match(:expr_call)
 match(:identifier)
end

rule :assignment do
 match(:expr_assignment)
 match(:type_assignment)
end

rule :expr_assignment do
 match(:identifier, '=', :expr) { |i, _, e| AssignmentNode.new(i, e) }
 match('^', :identifier, '=', :expr) do |_, i, _, e|
  AssignmentNode.new(i, e, true)
 end
end

rule :type_assignment do
 match(:identifier, :type_dec) { |i, td| AssignmentNode.new(i, td) }
end

rule :return do
 match('<-', :expr) { |_, e| InterruptNode.new(:return, e) }
 match('<-') { InterruptNode.new(:return) }
end

rule :type do
 match(:bool)
 match(:int)
 match(:float)
 match(:string)
```

```
  match(:array)
  match(:hash)
  match(:nil)
end

rule :hash_arg_list do
  match(:hash_arg, ',', :hash_arg_list) { |s, _, l| [s].concat(l) }
  match(:hash_arg) { |x| [x] }
end

rule :hash_arg do
  match(:value, ':', :value) { |lhs, _, rhs| [lhs, rhs] }
end

rule :hash do
  match('{', :hash_arg_list, '}') { |_, h, _| HashNode.new(h) }
  match('{', '}') { HashNode.new }
end

rule :array do
  match('[', :arg_list, ']') { |_, al| ArrayNode.new(al) }
  match('[', ']') { ArrayNode.new([]) }
end

rule :string do
  # Remove quotes around string
  match(/"[^"]*"/) do |s|
    ConstantNode.new(s.length <= 2 ? '' : s[1...-1])
  end
end

rule :float do
  match(Float) { |f| ConstantNode.new(f) }
end

rule :int do
  match(Integer) { |i| ConstantNode.new(i) }
end

rule :bool do
  match('true') { ConstantNode.new(true) }
```

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

    rule :nil do
      match('nil') { ConstantNode.new(nil) }
    end
  end
end

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

def parse(str)
  @shlp.parse str
end
end

if __FILE__ == $PROGRAM_NAME
sp = SHLParse.new
fail 'Usage: shlParse.rb <file>' if ARGV[0].nil?
f = File.read ARGV[0]
sp.log(false)
program = sp.parse f
program.evaluate
end
```

5.2 nodes.rb

```
require_relative './builtins'


class Scope
 attr_reader :vars
 def initialize(upper_scope = nil)
    @upper = upper_scope
    @vars = {}
    @callables = {}
 end


 def set_var(name, value, outer = false)
    if !outer then return @vars[name] = value
    elsif @upper.key?(name)
     return @upper.set_var(name, value)
    else
     return @upper.set_var(name, value, true)
    end
 end


 # recursively get variables through upper scopes if not found
 def get_var(name)
    if @vars.key?(name)
     result = @vars[name]
    elsif !@upper.nil?
     result = @upper.get_var(name)
    else
     result = :notfound
    end
    result
 end


 def key?(name)
    @vars.key? name
 end
```

```
 def add_callable(name, node)
    @callables[name] = node
 end


 def get_callable(name)
    if @callables.key?(name)
     result = @callables[name]
    elsif !@upper.nil?
     result = @upper.get_callable(name)
    else
     result = nil
    end
    result
 end
end

# Top level node, containing all other nodes.
class SHLProgramNode
 def initialize(statements)
    @statements = statements
 end


 def evaluate
    scope = Scope.new
    @statements.each do |s|
     r = s.evaluate(scope)
     if [:break, :continue].include? r[0]
       fail "Unexpected keyword \"#{r[0].to_s}\""
     end
     return r[1] if r[0] == :return
    end
 end
end


# Node for a block of code within brackets.
class BlockNode < SHLProgramNode
 def evaluate(scope)
```

```
    new_scope = Scope.new(scope)
    @statements.each do |s|
     r = s.evaluate(new_scope)
     return r if [:break, :continue, :return].include? r[0]
    end
  end

  def get_class_scope(scope)
    new_scope = Scope.new(scope)
    @statements.each do |s|
     r = s.evaluate(new_scope)
     if [:break, :return, :continue].include? r[0]
       fail "Unexpected keyword \"#{r[0].to_s}\""
     end
    end
    new_scope
  end
end

# Node for callable definition
class CallableDefNode
  def initialize(name, type, vars, block)
    @name, @type, @vars, @block = name, type, vars, block
  end

  def evaluate(scope)
    scope.add_callable(@name, CallableNode.new(@name, @type, @vars, @block))
    [:ok, nil]
  end
end

# callable node stored in callables
class CallableNode
  def initialize(name, type, vars, block)
    @name, @type, @vars, @block = name, type, vars, block
  end
```

```
 def evaluate(scope, params)
    new_scope = Scope.new(scope)

    # deep copy of vars to preserve values.
    vars_copy = []
    @vars.each { |e| vars_copy.push(e.dup) }

    # add param values to vars
    params.each_with_index { |p, i| vars_copy[i][1] = p }
    # check for :nv
    vars_copy.each { |v| puts 'Error: unassigned parameter' if v[1] == :nv }
    # add vars as variables to new scope
    vars_copy.each { |v| new_scope.set_var(v[0], v[1].evaluate(scope)[1]) }
    if @type == :func
     r = @block.evaluate(new_scope)
     r[0] = :ok if r[0] == :return
    elsif @type == :class
     r = [:ok, @block.get_class_scope(new_scope)]
    end
    r
 end
end

# Node for function calls.
class CallNode
 def initialize(node, params)
    @node, @params = node, params
 end

 def evaluate(scope)
    params = @params.inject([]) { |a, e| a << e.evaluate(scope)[1] }
    if @node.class == VariableNode && Builtins::General.key?(@node.name)
     return [:ok, Builtins::General[@node.name].call(*params)]
    end

    if @node.is_a? MemberNode
```

```
    n = @node
    s = scope
    while n.member.class == MemberNode
      s = s.get_var(n.instance.name)
      n = n.member
    end
    if [ConstantNode, ArrayNode, HashNode].include? n.instance.class
      return call_builtin(n, params, s)
    elsif n.instance.is_a? VariableNode
      val = n.instance.evaluate(s)[1]
      unless val.is_a? Scope
        val = MemberNode.new(ConstantNode.new(val), n.member)
        return call_builtin(val, params, s)
      end
    end
   end
   if @node.class == MemberNode
    scope = scope.get_var(@node.instance.name)
    while @node.member.class == MemberNode
      scope = scope.get_var(@node.name)
      @node = @node.member
    end
   end


   call = scope.get_callable(@node.name)
   if !call.nil?
    return call.evaluate(scope, @params)
   else
    fail "Error: no callable \"#{@node.name}\" found."
   end
 end
end

# Node for accessing a member in a class.
class MemberNode
 attr_reader :name, :instance, :member
```

```
 def initialize(instance, member)
   @instance, @member = instance, member
   if @member.is_a? MemberNode
    @name = member.instance.name
   else
    @name = member.name
   end
 end

 def evaluate(scope)
   if [ConstantNode, ArrayNode, HashNode].include? @instance.class
    return @instance.evaluate(scope)
   end
   instance_scope = scope.get_var(@instance.name)
   @member.evaluate(instance_scope)
 end

 def get_type(scope)
   scope.get_var(@instance.name).class
 end
end

# Node for a for loop.
class ForNode
 def initialize(comp, inc, block, assign = nil)
   @comp, @inc, @block, @assign = comp, inc, block, assign
 end

 def evaluate(scope)
   !@assign.nil? && @assign.evaluate(scope)

   while @comp.evaluate(scope)[1]
    new_scope = Scope.new(scope)
    r = @block.evaluate(new_scope)
    case r[0]
    when :continue
```

```
      next
    when :break
      return [:ok, nil]
    when :return
      return r
    end
    @inc.evaluate(scope)
   end
   [:ok, nil]
 end
end

# Node for a while loop.
class WhileNode
 def initialize(comp, block)
   @comp, @block = comp, block
 end


 def evaluate(scope)
   while @comp.evaluate(scope)
    new_scope = Scope.new(scope)
    r = @block.evaluate(new_scope)
    case r[0]
    when :continue
      next
    when :break
      return [:ok, nil]
    when :return
      return r
    end
   end
 end
end

# Node for return, break or continue
class InterruptNode
 def initialize(type, expr = nil)
```

```
    @type, @expr = type, expr
  end

  def evaluate(scope)
    val = @expr.nil? ? nil : @expr.evaluate(scope)[1]
    [@type, val]
  end
end

# Node for an if-statement
class ConditionalNode
  def initialize(i_block, ei_blocks = [], e_block = nil)
    @i_block = i_block
    @ei_blocks, @e_block = ei_blocks, e_block
  end

  def evaluate(scope)
    return @i_block.evaluate(scope) if @i_block.true?(scope)
    l = -> { !@e_block.nil? ? @e_block : nil }
    t = @ei_blocks.detect(l) { |ei| ei.true?(scope) }
    !t.nil? ? t.evaluate(scope) : [:ok, nil]
  end
end

# Holds the condition and the body of an ~i/~ei/~e
class IfNode
  def initialize(body, cond = ConstantNode.new(true))
    @body = body
    @cond = cond
  end

  def evaluate(scope)
    @body.evaluate scope
  end

  def true?(scope)
```

```
    @cond.evaluate(scope)[1]
  end
end

# Holds an expression which should evaluate to true or false
class ConditionNode
  def initialize(cond)
    @cond = cond
  end


  def evaluate(scope)
    [:ok, @cond.evaluate(scope)[1]]
  end
end

# Node for comparisons.
class ComparisonNode
  def initialize(lhs, rhs, op)
    @lhs, @rhs, @op = lhs, rhs, op
  end


  def evaluate(scope)
    [:ok, @lhs.evaluate(scope)[1].send(@op, @rhs.evaluate(scope)[1])]
  end
end

# Node for boolean operations
class BoolExprNode
  def initialize(lhs, rhs, op)
    @lhs, @rhs, @op = lhs, rhs, op
  end


  def evaluate(scope)
    lhs = @lhs.evaluate(scope)[1]
    rhs = @rhs.evaluate(scope)[1]
    result = @op == '&&' ? lhs && rhs : lhs || rhs
    [:ok, result]
```

```
 end
end

# Node for arithmetic operations.
class ArithmeticNode
 def initialize(lhs, rhs, op)
    @lhs, @rhs, @op = lhs, rhs, op
 end


 def evaluate(scope)
    lhs, rhs = @lhs.evaluate(scope)[1], @rhs.evaluate(scope)[1]
    case @op
    when '/'
     [:ok, lhs.to_f / rhs.to_f]
    when '//'
     [:ok, lhs / rhs]
    else
     [:ok, lhs.send(@op, rhs)]
    end
 end
end

# Node for assignment, stores a name and a value (node).
class AssignmentNode
 def initialize(var, value, outer = false)
    @var, @value, @outer, @c_scope = var, value, outer, nil
    if var.class == BracketCallNode
     @name = var
     @array = true
    else
     @name = var.name
    end
 end


 def evaluate(scope)
    if @var.class == MemberNode
     @var = get_correct_scope(@var, scope)
```

```
      @instance, @name = @var.instance, @var.name
      scope = scope.get_var(@instance.name)
    end
    if @array
      @name.set(scope, @value)
      [:ok, @value.evaluate(scope)[1]]
    else
      [:ok, scope.set_var(@name, @value.evaluate(scope)[1], @outer)]
    end
  end
end

# Node for getting the value from a variable.
class VariableNode
  attr_reader :name

  def initialize(name)
    @name = name
  end

  def evaluate(scope)
    value = scope.get_var(@name)
    if value != :notfound
      return [:ok, value]
    else
      fail "Error: no variable \"#{@name}\" found."
    end
  end
end

# Used when the bracket operator is called on an identifier
class BracketCallNode

  def initialize(identifier, arg)
    @identifier, @arg = identifier, arg
  end
```

```
 def evaluate(scope)
    arg_val = @arg.evaluate(scope)[1]
    var = @identifier.evaluate(scope)[1]
    val = var[arg_val]
    [:ok, val]
 end


 def set(scope, value)
    a = @identifier.evaluate(scope)[1]
    a[@arg.evaluate(scope)[1]] = value.evaluate(scope)[1]
 end
end

# Node for a hash.
class HashNode
 def initialize(args = nil)
    @args = args
    @hash = {}
 end


 def evaluate(scope)
    !@args.nil? && @args.each do |x|
     @hash[x[0].evaluate(scope)[1]] = x[1].evaluate(scope)[1]
    end
    [:ok, @hash]
 end
end

# For arithmetic operations with only one operand, such as "a++"
class UnaryExprNode
 def initialize(val, op, after, outer = false)
    @val, @op, @after, @outer = val, op, after, outer
 end


 def evaluate(scope)
    val = @val.evaluate(scope)[1]
    case @op
```

```
    when '-'
     return [:ok, -val]
    when '++', '--'
     scope.set_var(@val.name, val.send(@op[0], 1), @outer)
     return [:ok, @after ? val : @val.evaluate(scope)[1]]
    end
 end
end

# Node for negation expression e.g. !true
class NegationNode
 def initialize(val)
    @val = val
 end


 def evaluate(scope)
    [:ok, !@val.evaluate(scope)[1]]
 end
end

# Node representing an array.
class ArrayNode
 def initialize(array)
    @array = array
 end


 def evaluate(scope)
    return_array = []
    @array.each { |e| return_array << e.evaluate(scope)[1] }
    [:ok, return_array]
 end
end

# Node representing a cast
class ConversionNode
 def initialize(value, type)
    @value, @type = value, type
```

```
 end

 def evaluate(scope)
    val = @value.evaluate(scope)[1]
    case @type.evaluate(scope)[1]
    when String
     return [:ok, val.to_s]
    when Fixnum
     return [:ok, val.to_i]
    when Float
     return [:ok, val.to_f]
    when FalseClass
     return [:ok, !!val]
    end
 end
end

# Node representing a constant.
class ConstantNode
 def initialize(value)
    @value = value
 end

 def evaluate(scope)
    [:ok, @value]
 end
end
```

5.3 helpers.rb

```
require_relative './nodes'


# Function handle any if-statements with option ifelse and else
def if_statement_handler(if_body, if_cond, elseif = [], e = nil)
 if_node = IfNode.new(BlockNode.new(if_body), ConditionNode.new(if_cond))
 ConditionalNode.new(if_node, elseif, e.nil? ? e : IfNode.new(BlockNode.new(e)))
end


def for_statement_handler(body, assignment: nil, \
 cond: ConditionNode.new(ConstantNode.new(true)), inc: ConstantNode.new(nil))
 ForNode.new(cond, inc, BlockNode.new(body), assignment)
end


def call_builtin(var, params, scope)
 # If var is not a MemberNode, call is on a function in the global scope
 if var.class != MemberNode
   if Builtins::General.key?(var.name)
    return [:ok, Builtins::General[var.name].call(*params)]
   end
 end


 # Assume var is MemberNode
 val = var.instance.evaluate(scope)[1]
 c = var.evaluate(scope)[1].class
 hash = { String => Builtins::String,
          Array => Builtins::Array,
          Hash => Builtins::Hash,
          Fixnum => Builtins::Fixnum
         }[c]
 if !hash.nil? && hash.key?(var.member.name)
   return [:ok, hash[@node.member.name].call(val, *params)]
 end
 fail "Error: no method \"#{@node.member.name}\" for type \"#{c}\" found."
end
```

```
def get_correct_scope(var, scope)
 while var.member.class == MemberNode
    scope = scope.get_var(var.instance.name)
    var = var.member
 end
 var
end
```

5.4 builtins.rb

```
# Built in functions for strings
module StringFuncs
 def self.times(string, x)
    string * x
 end
end

# Built in function for arrays
module ArrayFuncs
 def self.includes(array, element)
    array.include? element
 end

 def self.append(array, element)
    array << element
 end
end

# Built in functions in the global scope
module GeneralFuncs
 def self.p(*args)
    print(*args)
 end

 def self.pl(*args)
    puts(*args)
 end
end

# Built in functions for hashes
module HashFuncs
 def self.has_key(hash, element)
    hash.key? element
 end
end
```

```
module IntegerFuncs
 def self.is_even(int)
    int.even?
 end
end

module Builtins
 General = { 'p' => GeneralFuncs.method(:p),
             'pl' => GeneralFuncs.method(:pl) }
 String = { 'times' => StringFuncs.method(:times) }
 Array = { 'includes' => ArrayFuncs.method(:includes),
           'append' => ArrayFuncs.method(:append)}
 Hash = { 'has_key' => HashFuncs.method(:has_key) }
 Fixnum = { 'is_even' => IntegerFuncs.method(:is_even) }
end
```