Ruby Language

# Grammar

## Microsyntax

INTEGER:   
[BEG] [1-9]([0-9\_]\*[1-9])?  
[BEG] ([0-7\_]\*[0-7])?  
[BEG] 0[xX][0-9a-fA-F]([0-9a-fA-F\_]\*[0-9a-fA-F])?  
[BEG] 0[dD][0-9]([0-9\_]\*[0-9])?  
[BEG] 0[bB][01]([01\_]\*[01])?  
[BEG] 0[oO][0-7]([0-7\_]\*[0-7])?  
[BEG] [?][:escaped-character:]

*escaped-character:*(\\(c|C-|M-))\*([^\\]|\\[abefnrstv]|\\[0-7]{1,3}|\\x[0-9a-fA-F]{1-2})

FLOAT:  
[BEG] (0|[1-9]([0-9\_]\*[0-9])?)[.][0-9\_]\*[0-9]([eE][+-]?[0-9]([0-9\_]\*[0-9])?)

GLOBAL-VARIABLE: (Thomas, p. 335)  
[BEG] [$][\_a-zA-Z][\_a-zA-Z0-9]\*  
[BEG] [$][~\_\\>=<;:/.,-\*$"!]  
[BEG] [$]-[0-9\_a-zA-Z]

NTH-REFERENCE:  
[BEG] [$][0-9]+

BACK-REFERENCE:  
[BEG] [$][`'+&]

ASSIGNMENT:  
[BEG] =  
[BEG] !=, %=, &=, \*=, +=, -=, /=, ^=, |=  
[BEG] ||=, &&=, <<=, >>=, \*\*=

CONSTANT:  
[BEG] v[A-Z][\_a-zA-Z0-9]\*

IDENTIFIER:  
[BEG] [\_a-z][\_a-zA-Z0-9]\*

FUNCTION-IDENTIFIER:  
[BEG] [\_a-zA-Z][\_a-zA-Z0-9]\*[!?=]? is = allowed in a call?

INSTANCE-VARIABLE:  
[BEG] @[\_a-zA-Z][\_a-zA-Z0-9]\*

CLASS-VARIABLE:  
[BEG] @@[\_a-zA-Z][\_a-zA-Z0-9]\*

REGULAR-EXPRESSION-BEGIN:  
[BEG] [/]|%r[:open:]

STRING-EMBEDDED-VARIABLE-BEGIN:  
[BEG] #  
[BEG; InString]

STRING-EMBEDDED-CODE-BEGIN:  
[BEG; InString] #[{]

WORDS-BEGIN:  
[BEG] %W[:open:]  
=> [BEG; InWords = Verbatim]

VERBATIM-WORDS-BEGIN:  
[BEG] %w[:open:]  
=> [BEG; InWords = Verbatim]

SHELL-STRING-BEGIN:  
[BEG] `  
[BEG] %x[:open:]  
=> [BEG; InString = ShellString]

U-MINUS-NUM:  
???

SEPARATING\_DOUBLE\_COLON: ‘::’S  
::

LEADING\_DOUBLE\_COLON: ‘::’L  
::

### Token Disambiguations

#### Disambiguation by local variable look-up

**Method call with array argument vs. array item access (CallAmbig.rb)**

IDENTIFIER whitespaceopt ‘[‘ arg ‘]’:

1. IDENTIFIER is local variable => array item access
2. whitespace => function call with parameter ‘[‘ arg ‘]’
3. array item access

**Here-doc vs. left shift (HeredocAmbig.rb)**

IDENTIFIER *whitespaceopt1* ‘<<’ *whitespaceopt2* (IDENTIFIER | *string*)

1. not *whitespace1* or *whitespace2* => left shift
2. IDENTIFIER is local variable => left shift
3. here-doc

## Production Rules

### Statements

program:   
[BEG] statement-list

statement-list:  
statementsopt terminatorsopt

statements:  
statement  
terminators statement   
statements terminators statement

statement:  
alias-statement  
expression-statement  
conditional-statement  
jump-statement  
statement rescue statement  
initializer  
finalizer  
undef undef-list

alias-statement:  
alias method-name-or-keyword [FNAME] method-name-or-keyword  
alias method-name-or-keyword [FNAME] symbol  
alias symbol [FNAME] method-name-or-keyword  
alias symbol [FNAME] symbol  
alias GLOBAL-VARIABLE GLOBAL-VARIABLE  
alias GLOBAL-VARIABLE BACK-REFERENCE

undef-list:  
undef-list ‘,’ [FNAME] method-name-or-keyword  
undef-list ‘,’ [FNAME] symbol  
method-name-or-keyword  
symbol

initializer:  
BEGIN ‘{‘ statement-list ‘}’

finalizer:  
END ‘{‘ statement-list ‘}’

conditional-statement:  
statement if expression  
statement unless expression  
statement while expression  
statement until expression

jump-statement:  
return open-argsopt moved from command-call and primary-expression, not an expression  
break open-argsopt moved from command-call and primary-expression, not an expression  
next open-argsopt moved from command-call and primary-expression, not an expression  
redo moved from primary-expression, not an expression  
retry moved from primary-expression, not an expression

expression-statement:  
expression  
assignment-expression-statement

assignment-expression-statement:  
lhs ‘=’ command-call  
compound-lhs ‘=’ command-call  
variable ‘=’ command-call  
lhs ‘=’ compound-rhs  
compound-lhs ‘=’ arg  
compound-lhs ‘=’ compound-rhs  
primary-expression ‘[‘ array-keyopt ‘]’ ASSIGNMENT command-call  
primary-expression ‘.’ IDENTIFIER ASSIGNMENT command-call  
primary-expression ‘::’S IDENTIFIER ASSIGNMENT command-call  
primary-expression ‘.’ CONSTANT ASSIGNMENT command-call  
back-ref ASSIGNMENT command-call

rescue-clauses:

rescue-clauses rescue-clause  
rescue-clause

rescue-clause:  
rescue exception-type-listopt exception-variableopt then statement-list

exception-type-list:  
arg  
m-rhs

exception-variable:  
‘=>’ lhs

ensure-clause:  
ensure statement-list

### Left Values

lhs:  
variable  
primary-expression ‘[‘ array-keyopt ‘]’  
primary-expression ‘.’ IDENTIFIER  
primary-expression ‘::’S IDENTIFIER  
primary-expression ‘.’ CONSTANT  
primary-expression ‘::’S CONSTANT  
‘::’L CONSTANT  
back-ref

compound-lhs-leaf: lhs duplicate due to ambiguity  
lhs

compound-lhs: defines a forest  
compound-lhs-head compound-lhs-nodeopt

compound-lhs-head compound-lhs-tailopt

L-PAREN compound-lhs ‘)’

compound-lhs-tail:  
STAR compound-lhs-leaf opt

compound-lhs-head:  
compound-lhs-head compound-lhs-node ‘,’  
compound-lhs-node ‘,’

compound-lhs-node:  
compound-lhs-leaf  
L-PAREN compound-lhs ‘)’

### Expressions

expression:  
arg   
command-call  
expression and expression  
expression or expression  
not expression  
‘!’ command-call why this special case here?

arg:  
primary-expression   
assignment-expression  
arg arg-binary-op arg  
arg-unary-prefix-op arg  
U-MINUS-NUM INTEGER ‘\*\*’ arg  
U-MINUS-NUM FLOAT ‘\*\*’ arg  
defined new-lineopt arg  
arg ‘?’ arg ‘:’ arg

command-call:  
command  
block-call  
block-call ‘.’ method-name command-args  
block-call ‘::’S method-name command-args

command:  
function-name command-args cmd-brace-blockoptprimary-expression ‘.’ method-name command-args cmd-brace-blockoptprimary-expression ‘::’S method-name command-args cmd-brace-blockoptsuper command-args  
yield command-args

cmd-brace-block:  
‘{‘ block-var-declopt statement-list ‘}’

primary-expression:  
numeric-literal  
symbolic-literal  
string-expression  
array-constructor

hash-constructor  
declaration-expression  
yield-expression  
block-expression

declaration-expression  
variable  
back-ref  
FUNCTION-IDENTIFIER  
primary-expression ‘::’S CONSTANT  
‘::’L CONSTANT  
primary-expression ‘[‘ array-keyopt ‘]’

function-name brace-block  
method-call  
method-call brace-block  
conditional-expression  
loop-expression

defined new-lineopt ‘(‘ expression ‘)’

string-expression:  
string-concatenation  
symbolic-expression  
shell-string  
regular-expression

assignment-expression:  
lhs ‘=’ arg  
lhs ‘=’ arg rescue arg  
variable ASSIGNMENT arg  
primary-expression ‘[‘ array-keyopt ‘]’ ASSIGNMENT arg  
primary-expression ‘.‘ IDENTIFIER ASSIGNMENT arg  
primary-expression ‘::‘S IDENTIFIER ASSIGNMENT arg  
primary-expression ‘.’ CONSTANT ASSIGNMENT arg  
back-ref ASSIGNMENT arg

yield-expression:  
yield ‘(‘ open-args ‘)’  
yield ‘(‘ ‘)’  
yield

conditional-expression:

if expression then statement-list if-tail end

unless expression then statement-list elseopt end  
case expressionopt terminatorsopt when-clausesopt elseopt end

loop-expression:

while **EXPECT-LOOP-DO[** expression do **]** statement-list end  
until **EXPECT-LOOP-DO[** expression do **]** statement-list end  
for block-var in **EXPECT-LOOP-DO[** expression do **]** statement-list end

command-args:  
[!CMD-ARG-PUSH(1)]  
open-args  
[END-ARG] ‘SPACE(‘ ‘)’ ... warning: do not put space before parentheses  
‘SPACE(‘ closed-args [END-ARG] ‘)’ ... warning: do not put space before parentheses  
[!CMD-ARG-POP]

parenthesized-args:  
‘(‘ ‘)’  
‘(‘ open-args new-lineopt ‘)’   
‘(‘ block-call new-lineopt ‘)’ ... warning: parenthesize args. for future version  
‘(‘ args ‘,’ block-call new-lineopt ‘)’ ... warning: parenthesize args. for future version

open-args:  
maplets block-argumentoptmaplets ‘,’ STAR arg block-argumentoptargs block-argumentopt  
args ‘,’ STAR arg block-argumentopt  
args ‘,’ maplets block-argumentoptargs ‘,’ maplets ‘,’ STAR arg block-argumentopt  
STAR arg block-argumentopt  
‘&’ arg  
command ... warning: parenthesize args. for future version

closed-args:  
maplets block-argumentopt  
maplets ‘,’ STAR arg block-argumentopt  
arg ‘,’ STAR args block-argumentopt  
arg block-argument  
arg ‘,’ STAR arg block-argumentopt  
arg ‘,’ args ‘,’ STAR arg block-argumentopt  
arg ‘,’ maplets block-argumentopt  
arg ‘,’ args ‘,’ maplets block-argumentopt  
arg ‘,’ maplets ‘,’ STAR arg block-argumentopt  
arg ‘,’ args ‘,’ maplets ‘,’ STAR arg block-argumentopt  
STAR arg block-argumentopt  
‘&’ arg

array-key:  
command new-lineopt ... warning: parenthesize args. for future version  
args traileropt  
args ‘,’ STAR arg new-lineopt  
maplet traileropt  
STAR arg new-lineopt

block-argument:  
‘,’ ‘&’ arg

args:  
arg  
args ‘,’ arg

compound-rhs:  
args ‘,’ arg  
args ‘,’ STAR arg  
STAR arg

then:  
terminator  
‘:’  
then  
terminator then

do:  
terminator  
‘:’  
LOOP-DO

if-tail:  
else-clauseopt  
elsif expression then statement-list if-tail

else-clause:  
else statement-list

block-var:  
lhs  
compound-lhs

opt-block-decl:  
empty  
‘|’ ‘|’  
‘||’  
‘|’ block-var ‘|’

block-call:  
command do-block  
block-call ‘.’ method-name parenthesized-argsoptblock-call ‘::’S method-name parenthesized-argsopt

method-call:  
function-name parenthesized-args  
primary-expression ‘.’ method-name parenthesized-argsopt  
primary-expression ‘::’S method-name parenthesized-args  
primary-expression ‘::’S non-constant-method-name  
super parenthesized-args  
super

brace-block:  
‘{‘ block-var-decl statement-list ‘}’  
DO block-var-decl statement-list end

do-block:  
BLOCK-DO block-var-declopt statement-list end

when-clauses:

when-clauses when-clause

when-clause

when-clause:  
when when-args then statement-list

when-args:  
args  
args ‘,’ STAR arg  
STAR arg

maplets:  
maplet  
maplets ‘,’ maplet

maplet:  
arg ‘=>’ arg

member-separator:  
‘.’  
‘::’S

new-line:  
‘\n’

trailer:  
NEW-LINE   
‘,’

terminator:  
NEW-LINE  
‘;’

terminators:  
terminator  
terminators ‘;’

### Block Expressions

block-expression:  
begin body end

‘SPACE(‘ expression [END-ARG] new-lineopt ‘)’ ... warning: ( ) as grouped expression  
‘(‘ statement-list ‘)’

declaration-expression:  
class qualified-module-name super-class body end  
class L-SHIFT expression terminator body end  
module qualified-module-name body end  
def method-name-or-keyword parameters-decl body end  
def singleton member-separator [FNAME] method-name-or-keyword [END]   
 parameters-decl body end

body:

statements rescue-clausesopt else-clauseopt ensure-clauseopt

super-class:  
terminator  
‘<’ [BEG] expression terminator

singleton:  
variable  
‘(‘ [BEG] expression new-lineopt ‘)’

### Formal Parameters

parameters-decl:  
‘(‘ parametersopt new-lineopt ‘)’ [BEG]  
parametersopt terminator

parameters:  
parameter-list ‘,’ default-parameter-list ‘,’ parameter-array block-parameteropt  
parameter-list ‘,’ default-parameter-list block-parameteropt  
parameter-list ‘,’ default-parameter-list block-parameteropt  
parameter-list ‘,’ block-parameteropt  
default-parameter-list ‘,’ parameter-array block-parameteropt  
default-parameter-list block-parameteropt  
parameter-array block-parameteropt  
block-parameter

parameter-list:  
parameter  
parameter-list ‘,’ parameter

parameter:  
IDENTIFIER

default-parameter-list:  
parameter ‘=’ arg  
default-parameter-list ‘,’ parameter ‘=’ arg

parameter-array:  
‘\*’ parameteropt  
STAR parameteropt

block-parameter:  
‘&’ parameteropt  
AMPERSAND parameteropt

### Literals and Strings

numeric-literal   
INTEGER  
FLOAT  
UNARY-MINUS-FOLLOWED-BY-NUMBER INTEGER  
UNARY-MINUS-FOLLOWED-BY-NUMBER FLOAT

symbolic-literal:  
SYMBOL-BEGIN method-name-or-keyword [END]  
SYMBOL-BEGIN variable-name [END]

symbolic-expression:  
SYMBOL-BEGIN string-contentsopt STRING-END

string-contents:  
string-contents string-content

string-content:  
STRING-CONTENT  
STRING-EMBEDDED-VARIABLE-BEGIN [BEG] embedded-variable  
STRING-EMBEDDED-CODE-BEGIN [BEG] [!COND-PUSH(0)] [!COND-PUSH(0)] statement-list ‘}’  
[!COND-LEX-POP] [!CMD-ARG-LEX-POP]

string-embedded-variable:  
GLOBAL-VARIABLE  
INSTANCE-VARIABLE  
CLASS-VARIABLE  
back-ref

string:  
STRING-BEGIN string-contentsopt STRING-END

string-concatenation:  
string  
string-concatenation string

shell-string:  
SHELL-STRING-BEGIN string-contentsopt STRING-END

regular-expression:  
REGULAR-EXPRESSION-BEGIN string-contentsopt REGULAR-EXPRESSION-END

### Array and Hash Table Constructors

array-constructor:

‘[‘ array-keyopt ‘]‘  
words  
verbatim-words

words:  
WORDS-BEGIN ‘ ‘ STRING-END  
WORDS-BEGIN word-listopt STRING-END

verbatim-words:  
VERBATIM-WORDS-BEGIN ‘ ‘ STRING-END  
VERBATIM-WORDS-BEGIN verbatim-word-listopt STRING-END

word-list:  
word-list string-contents ‘ ‘

verbatim-word-list:  
verbatim-word-list STRING-CONTENT ‘ ‘

hash-constructor:

‘{‘ mapletsopt traileropt ‘}’

‘{‘ argsopt traileropt ‘}’

### Variables

variable:  
IDENTIFIER  
variable-name  
CONSTANT  
nil  
self  
true  
false  
\_\_FILE\_\_  
\_\_LINE\_\_

back-ref:  
NTH-REFERENCE  
BACK-REFERENCE

### Operators

op: one of   
‘|’ ‘^’ ‘&’ ‘<>’ ‘==’ ‘===’ ‘~=’ ‘>’ ‘>=’ ‘<’ ‘<=’ ‘<<’ ‘>>’ ‘+’ ‘-‘ ‘\*’ STAR ‘/’ ‘%’ ‘\*\*’ ‘~’ ‘+@’ ‘-@‘ ‘[]’ ‘[]=’ ‘`’

arg-binary-op: one of   
‘..’ ‘...’ ‘+’ ‘-’ ‘\*’ ‘/’ ‘%’ ‘\*\*’ ‘|’ ‘^’ ‘&’ ‘<>’ ‘>’ ‘>=’ ‘<’ ‘<=’ ‘==’ ‘===’ ‘!=’ ‘~=’ ‘!~’ ‘>>’ ‘<<’ ‘&&’ ‘||’

arg-unary-prefix-op: one of   
 ‘!’ ‘~’ ‘+@’ ‘-@’

### Names

constant-name:  
CONSTANT

variable-name:  
GLOBAL-VARIABLE  
INSTANCE-VARIABLE  
CLASS-VARIABLE

function-name:  
FUNCTION-IDENTIFIER   
IDENTIFIER  
constant-name

method-name:  
function-name  
op [END]

method-name-or-keyword:  
method-name  
keyword [END]

non-constant-method-name:  
FUNCTION-IDENTIFIER  
IDENTIFIER  
op

qualified-module-name:  
‘::’L constant-name  
constant-name  
primary-expression ‘::’S constant-name

keyword: one of   
\_\_FILE\_\_ \_\_LINE\_\_ BEGIN END alias and begin break case class def defined? do else elsif end ensure false for if in module next nil not or redo rescue retry return self super then true undef unless until when while yield

## Runtime

### Control Flow Implementation

Stack Frames:

**Top Frame**

RFC defined (with InMethod flag == false), top-level lexical scope

**Primary Frames (Methods + Top Frame)**

RFC defined (with InMethod flag == true), lexical scope defined

**Secondary Frames**

1. eval frames, uses given lexical scope (binding)
2. block frames, lexical scope of the containing method

**RuntimeFlowControl**

created for every primary frame:

* InTryRescue – we are in any begin-rescue statements (starting from method body)
* InLoop – we are in any while/until loop (starting from method body)
* HasBlock – we are in a function that got a block
* InBlock – we are in any block (starting from method body) => the current code can be executed only using CallBlock (yield) - ! interop – delegate blocks need to be wrapped !
* IsAlive – the frame is alive
* InMethod – the frame is a method frame

**Stack Unwinders:**

* EvalUnwinder
  + unwinds evals (used by control flow ops in eval frames)
* MethodUnwinder
  + unwinds method frames (used by stack-unwinding control flow ops – return/break)

**Blocks:**

* Block
  + block is declared via BlockDefinition AST node (syntax: do ... end or { })
* Proc
  + a block becomes a proc when the block is passed to a method frame (whether or not & operator is used)
  + the method frame performing the conversion from block to proc is called ProcConverter
* Lambda
  + a lambda proc is created by *lambda* built-in function
  + the lambda function returns a copy of the given proc and marks it as Lambda proc
  + the *lambda*’s frame is a ProcConverter for the new block

#### retry

**semantics:**

|  |  |
| --- | --- |
| 1.  2.  3.  3.1.  3.2.  4.  4.1  4.2  4.3 | let Retry frame =  if frame.IsEval  # exit from eval() and do retry at call-site  frame’ = frame.Caller.Caller  Retry frame’  else if frame.Rescue ≠ nil then  # retry try-rescue block  frame.goto frame.Rescue.Block.Begin  else if frame.IsPrimary  if frame.Block ≠ nil then  **# unwind stack** until block passed differs from the currently passed block  # then repeat the call  frame’ = frame.Caller  if frame’.Block == frame.Block then  Retry frame’  else  frame’.goto before frame.CallSite  end  else  # exception not rescue-able within the current frame  frame’ = frame.Caller  frame’.throw LJE(“retry used out of rescue”)  end  else # frame.{IsProc, IsLambda}    if frame.CallSite == Yield then  # do retry in the frame that invoked yield  frame’ = frame.Caller  Retry frame’  else if frame.CallSite == Proc#call then   frame.throw LJE(“retry from proc-closure”)  end  end |

**implementation:**

switch (inner most scope)

{

1. eval top level: RFC.EvalRetry

if (flowControl.InBlock && blockFlowControl.CallerKind != Yield)  
 {  
 throw LocalJumpError(“retry from proc-closure”)  
 }  
  
 if (flowControl.InRescue || flowControl.InBlock || block != null)

{

throw new EvalUnwinder(RuntimeFlowControl.RetrySingleton, reason = Retry);

}

else

{

throw new LocalJumpError(“retry used out of rescue”, skipFrame = flowControl);

}

1. rescue clause

retrying = true;

continue;

1. a primary frame: RFC.MethodRetry

if (block != null)

{

return RuntimeFlowControl.RetrySingleton;

}

else

{

throw new LocalJumpError(”retry used out of rescue”, skipFrame = flowControl);

}

1. a block: RFC.BlockRetry

if (blockFlowControl.CallerKind == Yield)

{

blockFlowControl.ReturnReason = Retry;

return RuntimeFlowControl.RetrySingleton;

}   
else  
{

throw LocalJumpError(“retry from proc-closure”)

}

}

#### break <expression>

**semantics:**

|  |  |
| --- | --- |
| 1.      2.  3.  4.  4.1.  4.2  4.3. | let Break frame, value =  if frame.IsEval then  # exit from eval() and break at call-site  frame’ = frame.Caller.Caller  Break frame’, value  else if frame.Loop ≠ nil then   frame.Loop.Result = value  frame.goto frame.Loop.End  else if frame.IsPrimary then   # exception rescue-able within the current frame  frame.throw LocalJumpError(“unexpected break”)  else # frame.{ IsProc, IsLambda}  if frame.CallSite == Yield then  if frame.ProcConverter.IsActiveMethod then  **# unwinds stack** and returns the value from the ProcConverter  frame’ = frame.ProcConverter.Caller   frame.ProcConverter.Result = value   else  # exception not rescue-able within the current frame (block) frame’ = frame.Caller  frame’.throw LocalJumpError(“break from proc-closure”)  end  else if frame.CallSite == Proc#call then   if frame.ProcConverter.IsActiveMethod then  **# unwinds stack** and returns the value from the ProcConverter  frame’ = frame.ProcConverter.Caller  frame.ProcConverter.Result = value   else if frame.IsLambda then   # returns to the call operator, which returns to its caller  frame’ = frame.Caller.Caller  frame.Caller.Result = value   else  # exception not rescue-able within the current frame (block) frame’ = frame.Caller   frame’.throw LocalJumpError(“break from proc-closure”)  end  end  end |

**implementation:**

switch(inner most scope)

{

1. eval top level: RFC.EvalBreak

if (flowControl.InLoop || flowControl.InBlock)

{

throw new EvalUnwinder(<expression>, reason = Break,   
 targetFrame = blockFlowControl.Proc.Converter);

}

else

{

throw new LocalJumpError(“unexpected break”)

}

1. a loop

result = <expression>

break

1. a method: RFC.MethodBreak

<expression>  
throw new LocalJumpError(“unexpected break”)

1. a block: RFC.BlockBreak

blockFlowControl.TargetFrame = blockFlowControl.Proc.Converter  
blockFlowControl.ReturnReason = Break

return <expression>

}

#### next <expression>

**semantics:**

|  |  |
| --- | --- |
| 1.      2.  3.  4. | let Next frame, value =  if frame.IsEval then  # exit from eval() and do next at call-site  frame’ = frame.Caller.Caller  Next frame’, value  else if frame.Loop ≠ nil then   frame.goto frame.Loop.Condition.Begin  else if frame.IsPrimary then   # exception rescue-able within the current frame  frame’ = frame.Caller  frame’.throw LocalJumpError(“unexpected next”)  else # frame.{ IsProc, IsLambda}  # return to the yield/call’s caller:  frame’ = frame.Caller.Caller   # return value of a call to a lambda is nil, otherwise it’s the value passed to the next  if frame.IsLambda and frame.CallSite == Call then  frame.Caller.Result = nil  else  frame.Caller.Result = value  end  end |

**implementation:**

switch (inner most scope)

{

1. eval top level: RFC.EvalNext

if (flowControl.InLoop || flowControl.InBlock)

{

throw new EvalUnwinder(<expression>, reason = Next);

}

else

{

throw new LocalJumpError(“unexpected next” , skipFrame = flowControl)

}

1. a loop

<expression>

continue

1. a method: RFC.MethodNext  
     
   throw new LocalJumpError(“unexpected next”, skipFrame = flowControl)
2. a block:

return <expression>

}

#### redo

**semantics:**

|  |  |
| --- | --- |
| 1.      2.  3.  4. | let Redo frame =  if frame.IsEval then  # exit from eval() and do next at call-site  frame’ = frame.Caller.Caller  Redo frame’  else if frame.Loop ≠ nil then   frame.goto frame.Loop.Condition.End  else if frame.IsPrimary then   # exception not rescue-able within the current frame  frame’ = frame.Caller  frame’.throw LocalJumpError(“unexpected redo”)  else # frame.{ IsProc, IsLambda}  # local jump befor the first body statement (block arguments are not reassigned)  frame.goto frame.Body.Begin  end |

**implementation**:

switch (inner most scope)

{

1. eval top level: RFC.EvalRedo

if (flowControl.InLoop || flowControl.InBlock)

{

throw new BlockUnwinder(reason = Redo);

}

else

{

throw new LocalJumpError(“unexpected redo”)

}

1. a loop

redo = true  
goto loop.redo

1. a method: RFC.MethodRedo

throw LocalJumpError(“unexpected redo”, skipFrame = flowControl)

1. a block:

goto block.redo

}

#### return <expression>

**semantics:**

|  |  |
| --- | --- |
| 1.      2.  3.  3.1.  3.1.1.  3.1.2.  3.2.  3.2.1.  3.2.2.  3.2.3. | let Return frame, value =  if frame.IsEval then  # exit from eval() and do return at call-site  frame’ = frame.Caller.Caller  Return frame’, value  else if frame.IsPrimary then   # simple return:  frame’ = frame.Caller  frame.Result = value  else # frame.{IsProc, IsLambda}   if frame.Caller == Yield then  if frame.Owner.IsActiveMethod then  **# unwinds stack** and returns the value from the owner’s frame  frame’ = frame.Owner.Caller    frame.Owner.Result = value  else  # throws exception rescue-able in the current frame  frame.throw LocalJumpError(“unexpected return”)    end  else if frame.Caller == Proc#call then  if frame.IsLambda then  # return to the frame that called the *call* function:  frame’ = frame.Caller.Caller  frame.Caller.Result = value  else if frame.Owner.IsActiveMethod then  # returns the value from the owner’s frame  frame’ = frame.Owner.Caller  frame.Owner.Result = value    else  # throws exception rescue-able in the current frame  frame.throw LocalJumpError(“unexpected return”)  end   end  end |

**implementation:**

switch (inner most scope)

{

1. eval top-level: RFC.EvalReturn

value = <expression>

if (flowControl.InBlock)

{  
 if (blockFlowControl.CallerKind == Call && block.IsLambda)   
 {

throw new BlockUnwinder(reson = Next, value)

}

if (block.Owner.IsActiveMethod)   
 {

throw new MethodUnwinder(block.Owner, value)

}

throw new LocalJumpError("unexpected return")

}  
else  
{

throw new MethodUnwinder(flowControl, value)

}

1. a method

return <expression>

1. a block: RFC.BlockReturn

value = <expression>

if (blockFlowControl.CallerKind == Call && block.IsLambda)   
{

return value

}

if (block.Owner.IsActiveMethod)   
{

throw new MethodUnwinder(block.Owner, value)

}

throw new LocalJumpError("unexpected return")

}

#### yield <args>

BlockFlowControl bfc = BlockFlowControl.CreateForYield(block)

try

{

result = dynamic-site.Invoke(blockFlowControl, args);

}

catch (EvalUnwinder u)

{

result = blockFlowControl.GetUnwinderResult(u)

}

if (flowControl.{Block|Method|Eval}Yield(blockFlowControl, result)) return result;

result

// we can combine the cases into helpers:

// case “block”: if (flowControl.BlockYield(blockFlowControl, result)) return result;  
// case “primary frame”: if (flowControl.MethodYield(blockFlowControl, result)) return result;

// case “eval top level”: if (flowControl.EvalYield(blockFlowControl, result)) return result;

*Helpers:*

**BlockFlowControl∷CreateForYield: Proc → bool:**

if (block != null)

{

blockFlowControl = new BlockFlowControl(block, Yield)

}

else  
{

throw new LocalJumpError(“no block given”)

}

**BlockFlowControl∷GetUnwinderResult: EvalUnwinder → object:**

this.ReturnReason = u.Reason;

return u.ReturnedValue;

**RuntimeFlowControl∷{Block|Method|Eval}Yield: BlockFlowControl → object → object:**

switch (blockFlowControl.ReturnReason)

{

case Retry:

<retry>

case Break: // RFC.Yield{Block|Method|Eval}Break

if (blockFlowControl.TargetFrame.IsActiveMethod)  
 {

if (blockFlowControl.TargetFrame == flowControl)

{

<break> // recursive, but the same target frame

}

else  
 {  
 throw new MethodUnwinder(  
 targetFrame = blockFlowControl.TargetFrame,

returnValue = result

)

}

}  
 else  
 {  
 throw new LocalJumpError(“break from proc-closure”)  
 }

}

#### Proc#Call <args>

blockFlowControl = new BlockFlowControl(Call)

try

{

result = dynamic-site.Invoke(blockFlowControl, args)

}

catch (EvalUnwinder u)

{

result = blockFlowControl.GetUnwinderResult(u)

}

if (flowControl.MethodProcCall(blockFlowControl, result)) return result;

result

*Helpers:*

**RuntimeFlowControl∷MethodProcCall: BlockFlowControl → object → object:**

switch (blockFlowControl.ReturnReason)

{

case Retry:  
 Assert(false, “cannot retry block invoked via call”)

case Break:

if (blockFlowControl.Proc.IsLambda)

{

return result

}

else  
 {

// RFC.MethodYield

}  
}

#### Body with EH

**Implementation notes:**

Basic idea: All Ruby library functions that throw exceptions set $! just before the throw. Although we could implement library functions in the same way, exceptions thrown by non-Ruby CLR dlls would behave differently (with respect to $!). Therefore we need another approach. Instead of setting $! at throw time, we set it at capture time, i.e. just before the user code can actually see this side-effect of throw.

$! is saved and restore by rescue and ensure clauses. Although $! behaves like a stack, we don’t need explicit stack. We can use execution stack (store old values of $! into local variables). This optimization can be done because push/pop are paired within begin-rescue/ensure-end blocks.

(Possible opts: class/module/begin defs – do not have blocks; no eval)

bool retrying, exception\_thrown  
Exception rethrow, old\_$!

loop

{

retrying = false;

exception\_thrown = false;  
 rethrow = false

#if any rescue clause

old\_$! = GetCurrentException()   
 #endif

try

{

try

{

<guarded statements>

}

filter (Exception e, rfc.CanRescue(e))

{

exception\_thrown = true

#if any rescue clause

#if top-level

flowControl.InRescue = true // top-level only

#end

try

{

if (evaluate all expressions in 1st rescue; any of the results === $!)

{

<l-value-1> = $!

< statements-1>

}

else if ...

...

else if (evaluate all expressions in nth rescue; any of the results === $!)

{

<l-value-n> = $!

<statements-n>

}  
 else  
 {

rethrow = true  
 }

}

filter (EvalUnwinder u, u.Reason == Retry)

{

retrying = true

continue

}

#else

rethrow = true

#endif

}

#if any rescue clause

finally

{

if (!rethrow) SetCurrentException(old\_$!)

#if top-level

flowControl.InRescue = false  
 #endif

}  
 #endif

#if else clause

if (!exception\_thrown)

{

<else statements>

}  
 #endif

}

#if any rescue clause || any else clause

filter (Exception e, rfc.CanRescue(e))   
 {  
 rethrow = true  
 }  
 #endif

finally

{

if (!retrying)

{

#if ensure clause

old\_$! = GetCurrentException()

<ensured statements>

SetCurrentException(old\_$!) // can be skipped by jump

#endif

if (rethrow && old\_$! != null) throw old\_$!

}

}

break;

}

CanRescue(e) = not e is StackUnwinder || e is LJE and e.SkipFrame = this

#### All User methods

Used to unwind frames: frames are unwound to proc-converters or block owners. Since any user method can take a block and hence be a proc-converter, all methods needs to do this.

RuntimeFlowControl flowControl = CreateRfcForMethod(block)

try

{

<method body>

}

filter (Exception e; IsMethodUnwinderTargetFrame(scope, e))

{

return u.ReturnValue

}

finally

{

flowControl.IsActiveMethod = false

# interpreter: leave frame

}

*Helpers:*

IsMethodUnwinderTargetFrame: RubyScope → Exception → Bool

{

if u = e as MethodUnwinder != null

return u.FlowControl == flowControl

else

# capture stack trace

}

CreateRfcForMethod: Proc → RuntimeFlowControl!

{

result = new RuntimeFlowControl()

result.IsActiveMethod = true

if (proc != null && proc.Kind == Block)

{

Assert(proc.Converter == null)

proc.Converter = result  
 proc.Kind = Proc

}

}

#### Library Methods (Methods That Should Have Simple CF Structure)

Action binder: (helper method):

#if proc != null

bfc = BlockFlowControl∷CreateForLibraryMethod(proc)  
try

{

return <callee>(bfc, args)

}

filter (MethodUnwinder u, bfc.IsProcConverterTarget(u))

{

return u.ReturnValue

}

finally

{

bfc.LeaveProcConverter()  
}

#endif

*Helpers:*

BlockFlowControl ∷IsProcConverterTarget: MethodUnwinder! → bool

return IsProcConverter && u.TargetFrame == proc.Converter

BlockFlowControl ∷LeaveProcConverter: Proc! → void

if (proc.IsProcConverter) { proc.Converter.InActiveMethod = false; }

BlockFlowControl∷CreateForLibraryMethod : Proc! → BlockFlowControl!

if (proc.Kind == Block)

{

Assert(proc.Converter == null)

proc.Converter = new RuntimeFlowControl()  
 proc.Converter.IsActiveMethod = true  
 proc.Kind = Proc  
 isProcConverter = true

}  
else   
{

isProcConverter = false

}

return new BlockFlowControl(Yield, proc, isProcConverter)

Body (method not taking a block nor defining a block that can return)

if (blockFlowControl != null)

{

...

result = dynamic-site.Invoke(blockFlowControl, args)

if (blockFlowControl.BlockJumped(result)) return result

...

}

else

{

throw new LocalJumpError(“no block given”)

}

*Helper:*

BlockFlowControl∷BlockJumped: object → bool  
{  
 MethodYield(isProcConverter ? proc.Converter : null, bfc, returnValue)   
}

Body (method not taking a block and defining a block that can return)

TODO: flag → generate by binder

RFC rfc;

try

{

rfc = RFC.CreateForBlockOwner(blockFlowControl)

...

}

catch (MethodUnwinder u)   
{  
 if (u.TargetFrame == rfc) { return u.ReturnValue } else { throw }  
}  
finally  
{  
 rfc.InActiveMethod = false  
}

*Helper:*

RuntimeFlowControl∷CreateForBlockOwner: BlockFlowControl → RuntimeFlowControl  
{  
 if (bfc != null && bfc.IsProcConverter)  
 {  
 result = bfc.Proc.Converter;  
 }  
 else  
 {  
 result = new RuntimeFlowControl()  
 result.IsActiveMethod = true  
 }  
}

#### All Blocks

try {

var-expr-1 = arg-1  
...

var-expr-N = arg-N  
  
block.redo:

loop

{  
 try

{

<body>  
 return <body result>

}

catch (BlockUnwinder u)   
 {

if (u.IsRedo) continue  
 return u.ReturnValue

}

}

} filter (e: RubyOps.FilterBlockException(e)) {

# empty

} finally {

# interpreter: leave interpreted frame

}

#### A method call with a block

object result;

Proc proc;

**BLOCK DEF:** proc = RubyOps.DefineBlock(rfc, <block-delegate>)  
**BLOCK REF:** proc = <block-value>  
  
loop

{

try

{

**BLOCK DEF:** proc.Kind = Block

result = <method>(<args>, proc);

}  
 catch (EvalUnwinder u)  
 {

result = u.ReturnValue

}

if (result != RuntimeFlowControl.RetrySingleton) break;

if (block == proc)

{

<retry> // return result; repeat try-block, etc.

}

}

*Helpers:*

MethodUnwinder∷GetMethodCallWithBlockResult: MethodUnwinder → Proc → object:

if (u.TargetFrame == proc.Converter)

{

return this.ReturnValue

}

else

{

throw this

}

EvalUnwinder∷GetMethodCallWithBlockResult: EvalUnwinder → Proc → object:

// Retry and Break (break from a library function taking a block)

return this.ReturnValue

StackUnwinder∷GetMethodCallWithBlockResult: StackUnwinder → Proc → object:  
throw this

#### While/until <expression> do <statements> end

result = null;

try

{

flowControl.InLoop = true // top-level only

redo = false

loop  
 {

try

{  
 if (redo)  
 {  
 redo = false  
 }  
 else  
 {

**WHILE:** if (!<expression>) break

**UNTIL:** if (<expression>) break

}

<statements>

}

catch (BlockUnwinder u) // Redo, Next

{

redo = (u.Reason == Redo)

}

filter (EvalUnwinder u, u.Reason == Break)   
 {

result = u.ReturnValue

break

}

}

}

finally // top level only

{

flowControl.InLoop = false; // top level only

}

### Top-level Lexical Scope

1. In the entry-point script executed by MRI the top lexical scope is the Object class:

class Object

self = class << Object.new; <main singleton trait>; self; end

TOPLEVEL\_BINDING = binding

private

<code>

end

The Object is not included in Module.nesting.

1. In a hosted IronRuby scenario this is the model for top-level scope:

class Object

self = class << Object.new

<main singleton trait>

def method\_missing name

lookup dlr-scope

end

def TOPLEVEL\_BINDING

dlr-scope.Binding

end

self

end

dlr-scope.Binding = binding

private

<code>

end

The Object is not included in Module.nesting.

1. Kernel#load executes the code in an isolated mode. The top-level lexical scope is a module scope of a new module. Top level run-time scope (TOPLEVEL\_BINDING) is still available and the same. Self is a new *main* singleton.

module <anonymous>

self = class << Object.new

include <anonymous>

<main singleton trait>

self

end

private

<file content>

end

The anonymous module is included in Module#nesting.

### Class hierarchy



[MethodLookup.vsd](http://devdiv/sites/CLR/Merlin/Shared%20Documents/Ruby/MethodLookup.vsd)

* Rectangles represent class objects.
* ‘s’ arrow points from an object to its singleton class object
* ‘b’ arrow points from a class to its base class
  + Singleton classes are sealed (user classes cannot derive from them).
  + Class class is sealed.
  + Only classes and their singletons have an incoming b-edge.
  + A *module-function* method can only be defined on modules, not on classes.
* ‘c’ arrow points from an object to its class unless the object has a singleton.
* S(x) denotes the singleton class object for x
* S*i*(x) denotes the i-th singleton class object for x
* SD(x) denotes ‘dummy singleton’ for x – a terminator that s-points to itself
* Whenever a new module/class is defined 2 class objects are created: The module/class itself (e.g. module M/class C) and its singleton S(M)/S(C). At this point S(M) == SD(M), S(C) == SD(C).
* Whenever a singleton class is defined for an object *x*, it is checked whether there is already a singleton class for that object. If so, the existing singleton is returned. If not and the object
  + is a module or class

A new SD(x) is appended to the s-list and the previous one becomes S(x).

* + otherwise

S(x) class is created and s-relationship is established x -> S(x) == SD(x) via instance data of the object.

**Dummy singletons**

x → S(x) → SD(x)

DS(x) shares tables with S(x) until there is write operation on these tables.

At that very moment SD(x) is converted to S(S(x)) with new “empty” tables (tables containing default members for the singleton) and a new SD(S(x)) is created. We consider this behavior an implementation detail and don’t support it. New singletons in IronRuby can be created only explicitly using ≪ operator. No method sharing takes place, SD(x) is initialized as any other class.

#### Method Access

Read: an invocation of a method ‘f’ on object ‘x’:

1. S(x) defined?
   1. yes => follow S(x) once
   2. no (the object doesn’t have a singleton class object) => follow C(x) once
2. Follow B(x) until the method is found or Object is reached.
3. If not found search for ‘method\_missing’ in the same way

Write: a definition of a method ‘f’:

1. Instance (unqualified)
2. Singleton (qualified)

#### Super call

**Parameters**

TODO …

**Target method**

* Let **s** be the inner-most scope in the lexical scope hierarchy of the ‘super’ call site that is either
  1. a method scope, or
  2. a block definition scope of a block whose caller is a method created by ‘define\_method’
* If **s** is a method definition scope (a) then this method’s frame **f** is available: either it is
  1. the frame that invokes the ‘super’ call, or
  2. captured by a closure of the block that contains the ‘super’ call
* If **s** is a block (b) then its caller’s frame **f** is available.

Use the arguments, self object and method name of the frame **f** for the ‘super’ invocation.

Implementation: the method defined by define\_method passes its frame into the block it calls (in BlockParam object). Super call site traverses lexical hierarchy and finds scope **s**.

#### Constant Access

Let M is the module/class/singleton defined by the lexical scope inner-most to the constant access.

If the top-level scope the code was executed in isolated mode (Kernel#load wrap = true), let G be the anonymous module created by this method. Otherwise let G be the Object class.

Read:

1. Global constant read (::C)

Look at the constant table of **Object** and its ancestors.

1. Unqualified constant read (C)
2. Look at the constant table of each module/class/singleton whose definition is in the current lexical scope chain. Start from the inner most lexical, where the constant access takes place.  
   (Note: do not look at tables of ancestors of the modules).
3. Look at the constant table of the module **G**.  
   (Note: do not look at tables of ancestors of the modules).
4. If G != Object, look into Object’s constant table.
5. Look at constant tables of M’s ancestors in the ancestor hierarchy order.
6. Qualified constant read (expression::C)

Look at the constant table of the module <expression> and its ancestors.

Write:

1. Global constant write (::C)

Write to the constant table of **Object**.

Ruby 1.8 has very weird behavior for an assignment to ::C. It seems that it uses the current self.

Ruby 1.9: as above.

1. Unqualified constant write (C)
2. Find the inner-most module/class/singleton definition in the current lexical scope chain. Start from the inner most lexical, where the constant write takes place.
3. If no module found then write to the constant table of **G**.
4. Qualified constant write (expression::C)

Write to the constant table of the <expression> module.

#### Ancestors

1. “include m1, m2” is equivalent to “include m2; include m1”
2. Including module m1 which has not yet been included causes all of the
   1. Iterating over m1.ancestors from back to front, recursively include each ancestor module.
3. Including module m

### Generics

### Parallel Assignment

Parallel assignment semantics is used in a parallel assignment expression, a yield to a block, jump statements (return value) and an array constructor.

**Notation**:

R/L(n, -) ... right/left hand side: x1, ..., xn

R/L(n, \*) ... right/left hand side: x1, ..., xn, \*y

L(1+, -) ... left hand side: x1,

R/L[n] ... n-th value of RHS/LHS

R/L[\*] ... splatted value from RHS/LHS

**Rules**:

L(1+, -) is equivalent to L(2, -) with a placeholder as a second parameter.

Nested left values are handled recursively.

Nested L(1, -) are handled by an immediate recursion (i.e. removing parenthesis, w/o evaluating the current level) .

|  |  |  |  |
| --- | --- | --- | --- |
| LHS | RHS | result expression | writes |
| (1, -) | (0, \*) | result = Splat(R[\*]) | L[1] = result; |
| (1,\*) | result = SplatPair(R[0], R[\*]) |
| otherwise | result = MakeArray(RHS) |
| otherwise | (0, \*) | result = Unsplat(R[\*]) | ∀i: L[i] = result[i]  if (s == \*) L[\*] = result[n →] |
| (1, -) | result = Unsplat(R[1]) |
| otherwise | result = MakeArray(RHS) |

where runtime operations are:

Splat(obj) =

if obj is array then

if |obj| == 0 then nil else if |obj| == 1 then obj[1] else obj end

else

obj

end

SplatPair(obj, splattee) = if splattee == [] then obj else SplatAppend([obj], splattee) end

Unsplat(obj) = if obj is array then obj else [obj] end

MakeArray(RHS) = if RHS ~ R(n, \*) then SplatAppend([R[1] ... R[n]], R[\*]) else [R[1] ... R[n]] end

SplatAppend(a, s) = if s is array then a.AddRange(s) else a.Add(s) end

array[n →] ... gets slice of the array starting with element #n

array[n] ... tries to get an element #n, nil if out of range

**Other Forms**:

|  |  |  |  |
| --- | --- | --- | --- |
|  | LHS | RHS | implementation |
| block call | { |LHS| ... } | *yield* RHS | ~ LHS = RH, but warning for cases:  L(1,-) = R(n ≠ 1, -)  L(1,-) = R(n, \*) and n + |R[\*]| ≠ 1 |
| array construction | *result* | [RHS] | ~ L(1,-) = RHS,  but R(1,\*) → MakeArray(RHS) but using Unsplat not Splat |
| jump statement result | *Result* | *jump* RHS | ~ L(1,-) = RHS,  but R(1,\*) → MakeArray(RHS) |

### Mutable String

Basic architecture:

MutableString holds on Content and Encoding. Content is an abstract class that has three subclasses:

1. StringContent

* Holds on an instance of System.String – an immutable .NET string. This is the default representation for strings coming from CLR methods and for Ruby string literals.
* A textual write operation on the mutable string that has this content representation will cause implicit conversion of the representation to StringBuilderContent.
* A binary read/write operation triggers a transition to BinaryContent using the Encoding stored on the owning MutableString.

1. StringBuilderContent

* Holds on an instance of System.Text.StringBuilder – a mutable Unicode string.
* A binary read/write operation transforms the content to BinaryContent representation.
* StringBuilder is not optimal for some operations (requires unnecessary copying), we may consider to replace it with resizable char[].

1. BinaryContent

* A textual read/write operation transforms the content to StringBuilderContent representation.
* List<byte> is currently used, but it doesn’t fit many operations very well. We should replace it by resizable byte[].

The content representation is changed based upon operations that are performed on the mutable string. There is currently no limit on number of content type switches, so if one alternates binary and textual operations the conversion will take place for each one of them. Although this shouldn’t be a common case we may consider to add some counters and keep the representation binary/textual based upon their values.

The design assumes that the nature of operations implemented by library methods is of two kinds: textual and binary. And that data that are once treated as text are not usually treated as raw binary data later. Any text in the IronRuby runtime is represented as a sequence of 16bit Unicode characters (standard .NET representation). Each binary data treated as text is converted to this representation, regardless of the encoding used for storage representation in the file. The encoding is remembered in the MutableString instance and the original representation could be always recreated. Not all Unicode characters fit into 16 bits, therefore some exotic ones are represented by multiple characters (surrogates). If there is such a character in the string, some operations (e.g. indexing) might not be precise anymore – the n-th item in the char[] isn’t the n-th Unicode character in the string (there might be escape characters). We believe this impreciseness is not a real world issue and is worth performance gain and implementation simplicity.

### Loader

Implements Kernel#load\_assembly(assembly, type).

1. CLR assembly qualified type name -> load library, run initializer
2. CLR long assembly name -> load assembly

Implements Kernel#require and Kernel#load methods.

Argument *path* is

1. CLR long assembly name or qualified type name

* Load .NET assembly or C# implemented Ruby library if the type of the library initializer is specified. The library initializer needs to derive from LibraryInitializer class and is called to load the library classes and modules.
* The path is matched against pattern (<type-name>,)? <simple-assembly-name> (, <property-name>=<property-value>)+, property names are arbitrary identifiers and property values arbitrary strings not containing ‘=’ or ‘,’. Exact CLR property names are not matched.

1. Absolute path  
   switch (extension):
   * .rb -> load
   * .dll
     + MRI: load dll
     + IR: not supported
   * None
     + MRI: try load *path* + .rb, *path* + .dll
     + IR: try load *path* + *extension* for all registered extensions in runtime (.rb, .py, .js, …)
       - If multiple files match an ambiguous match exception is thrown.
2. Relative path

* For each *dir* in *LoadPaths*: Apply (2) on *dir* combined with *path*

Notes:

* Require checks $” variable for loaded files and appends the name of the loaded file with extension to it on success.
* Load doesn’t try to append extensions.

### Exceptions

Issues with dynamic frames:

* library functions: we would need try-fault blocks in rules for all lib functions that could possibly be on stack (not only taking a block, but also using a block passed earlier like hash.default, etc.)
* stack trace ends with the catch/filter that catches the exception
  + fault blocks above are not executed
* requires #line updates
* requires trace combination on rethrow

Issues with regular new StackTrace(exception):

* not available on Silverlight
* doesn’t provide full stack trace, the trace ends by the frame that catches the exception

Current solution:

* at catch block in rescue clause, we snapshot the current full stack trace and append it to the exception’s trace
* exceptions could still be thrown by “throw”, no special helper needed, CLR methods that throws behave the same as Ruby lib methods for Ruby user

## Compiler

### Lexical Scopes



### Optimizations

**No aliasing of binding/eval**

1. Don’t need to allocate a dictionary in inner-most variable scopes (method, block, module).

def closure\_binding

p = []

2.times { |i| # creates a local variable dictionary (scope)

p << binding # captures the scope

}

eval('puts i', p[0]) # 0

eval('puts i', p[1]) # 1

end

TODO:Methods that use scope vs. execution context.

# Interop Notes

Scenarios:

1. Hosting API interop

var scope = runtime.CreateScope(“rb”);

a)

scope.SetVariable(“x”, 1);

scope.SetVariable(“y”, 2);  
scope.Execute(“z = x + y”)

assert(scope.GetVariable(“z”) == 3)

Issues:

* locals cannot be live (tokenizer needs to know upfront)

scope.SetVariable(“x”, 1)

cc = rb.Compile(“puts x + 1”)

cc.Execute(scope)

scope.DeleteVariable(“x”)

// doesn’t fire method\_missing

cc.Execute(scope)

* metadata could help – only variables listed in metadata become local

class StrongScope

public int x;

end

meta = new Metadata(typeof(StrongScope))

cc = rb.Compile(“puts x + y”, meta)

scope1 = rb.CreateScope(meta)

scope1.SetVariable(“y”, 2)

// gets x directly from scope,

// y resolved via method\_missing on scope singleton

cc.Execute(scope1)

// error, cannot delete CLR field (maybe we can hide it)

scope1.DeleteVariable(“x”)

scope1.DeleteVariable(“y”)

// fires method\_missing

cc.Execute(scope1)

scope2 = rb.CreateScope(meta)

scope2.SetVariable(“y”, 4)

scope2.SetVariable(“puts”, “bar”)

// resolves y in scope via method\_missing,

// uses Kernel#puts since method\_missing is fired only

// if the method isn’t found in hierarchy

cc.Execute(scope2)

b)

scope = runtime.CreateScope()

scope.Execute(“x = 1”)

scope.Execute(“puts x”) // code is eval’d against the binding

c)

scope.Execute(@“  
 class << self

def foo; end

class C; end

end

def bar; end

class D; end

”)

// public method on scope singleton:

assert(scope.GetVariable(“foo”) != null);

// public class/module on scope singleton:

assert(scope.GetVariable(“C”) != null);

// method bar is private on Object

assert(scope.GetVariable(“bar”) == null);

// class D is on Object

assert(scope.GetVariable(“D”) == null);

Idea1:

* Ruby top-level binding is Scope content?
* Whatever is in the Scope at the point of the compilation is considered a local variable?  
  -> NO, this is why not:
* But we don’t have scope at compilation time (Compile method).

c1 = rb.Compile(“y = x + 1”) #1  
c2 = rb.Compile(“z = y<<2”) // y **is not** local here  
scope1.SetVariable(“x”, 1)

c1.Execute(scope1)  
c2.Execute(scope1)

vs.

meta = new Metadata()  
c1 = rb.Compile(“y = x + 1”, meta)  
c2 = rb.Compile(“z = y<<2”, meta) // y **is** local here  
scope1 = meta.CreateScope()

scope1.SetVariable(“x”, 1)

c1.Execute(scope1)  
c2.Execute(scope2)

vs.

scope1.SetVariable(“x”, 1)

scope1.Execute(“y = x + 1”) // x is (not) local here (see Note1)  
scope1.Execute(“z = y<<2”) // y **is** local here

(Note1)

* + It would be local if we loaded Metadata for compilation with the content of the scope.
  + It would require **name mangling**.
  + Could methods/classes be then available via the scope? **No**, **because the method names would be converted to locals** by the next Execute and would never be callable again:

scope1.Execute(“def self.foo; 1; end;”)

// foo is not in scope -> it becomes a local here:  
scope1.Execute(“puts foo”) # nil

* + Therefore, flowing live scope data into metadata doesn’t work.
  + But the scenario should work: “x” is resolved via *method\_missing* on main singleton.

Idea1.1:

Main issue of Idea1: inability to distinguish between a local stored in the scope and a method/class in the scope.

* Fix: Methods and classes are wrapped in an internal type; only non-methods and non-classes are loaded to the metadata.
* Still have an issue with
  + **name mangling** and
  + **undefined locals (#1) in compilation time** => the compiler emits method call
    - behavior would be different if first compiled and then executed or executed right away, we don’t want that.

Idea2:

* By default there is no mapping of locals to the values in the scope.
* Scope holds on to Ruby binding to support scenario (b) – transition of binding state across multiple executions. The tokenizer loads list of static locals from Metadata/CompilerOptions (newly introduced variables from previous executions could be transferred via CompilerOptions ~ Python’s TrueDivision/WithStatement flags).
* The static locals list is a part of the binding object; we can evaluate any code against the binding. It is the same as engine.Execute against the Scope that holds on the binding.
* Static locals could be mangled – if the name of the local is FooBar, the local variable will be foo\_bar.
* Scope is associated with main singleton that defines:
  + method\_missing
    - checks the scope for the name + try name mangling
    - found & is RubyMethod: IDO => invoke (wraps methods)
    - found & otherwise => return it
    - not found => call method\_missing.super
  + const\_missing
    - Unfortunately, this doesn’t work because constants are resolved lexically and the owner of the constant on top level is Object class. This only limits direct reference of capital cased variables from scopes as constants. The can still be retrieved by calling capital cased method or via name mangling.
* *class C*, *module M, public def m* on main singleton sets alias into the scope
  + alias => could get out of sync?
  + Is that a problem?
    - *remove\_method* -> *method\_removed* removes the method from the scope (only if the variable in the scope is actually a method).
    - *undef* method -> this is fine, since undef sets undefined-method dummy
    - host can remove methods that were added by *def m* -> it’s ok, calls to such methods will not even get to method\_missing, they will resolve by regular Ruby resolution order
    - re-def will rewrite content of the scope slot
    - perf: regular method calls get cached, only when method\_missing on singleton is hit no caching is possible (since we cannot guarantee the host didn’t update the scope storage)



Rb.Compile(@“  
 class << self

def foo; end

def foo=; end

end

// z is local, if z is defined local in the static binding

// we will write to the scope storage.

// If it is not defined it is written into Ruby binding, which

// should probably be writing into the scope.

z = 1

// method\_missing ‘x=’ -> writes value to the scope

self.x = 1

”).Execute(scope)

// x needs to be added by the code execution to make this working

Scope.GetVariable(“z”)

* expando scope – scope extensions are called for variable\_missing… -> no, problem with order (maybe arbitrary order + exception on duplicate – ambiguous member)
  + host could have go to dynamic binding as a fallback

Scope operations:

* scope extensions participate:
  + Ruby read: reads from dynamic binding if it was compiled as local, otherwise method\_missing lookup in the scope
  + Ruby write: always compiled as local, writes to dynamic binding
  + host read: get from host dict, if not found call all extensions, exception on duplicate or combination of duplicates; exception on not found
  + host write: writes to host dict, cannot write to dynamic binding – variables needs to be known upfront
* scope variables
  + Ruby read: the same
  + Ruby write: always compiled as local, writes to dynamic binding, propagates the write into scope – like methods and class on singleton

Problems?

1. Exec(“x = 1”) -> adds variable x to the scope
2. Exec(“x()”) -> this is ok, x was defined above as a local variable, so it will be local here also => this will fail correctly
3. SetVar(“x”, 2) -> sets variable in the scope
4. Exec(“puts x”) -> needs to get the updated value

(2) works because **local variable definition has higher priority than method call**.

(4) works if the local in dynamic binding is an alias for the value in the scope => the local in the binding are be wrapped to *RubyScopeVariable*. Needs to be unwrapped: we know which variables are dynamic – only those loaded from CompilerOptions/Metadata could be backed by scope.

* + Host read:
  + Host write:

# References

1. Thomas, D. *Programming Ruby - The Pragmatic Programmer's Guide, second edition.*