Compilers

HAL

Héctor Ramón Jiménez

Alvaro Espuña Buxo

January 11, 2015

Facultat d'Informàtica de Barcelona

Contents

1 Introduction

Our main objective was to create a programming language we would find **useful** in the future. With that in mind we have developed `HAL`, an scripting language with these characteristics.

- A **clean** syntax, perfect for creating **D**omain-**S**pecific **L**anguages
- A **consistent** object-oriented architecture with **inheritance**
- **Dynamic typing** and **duck typing**
- Builtin methods that can be **rewritten** in `HAL` itself
- **Module** imports
- **First-class** ~~functions~~ methods
- An **extensible**, **intuitive** and **interactive** interpreter

We use `Python` and `Ruby` a lot, and find them very interesting. In fact, most of the features of `HAL` are directly influenced by these two languages. However, we implemented everything in our own way, thinking in our needs as programmers and what made sense to include in `HAL`.

Thourough this document ther will be a lot of examples of `HAL`. The code will be highlighted and in monospace font. The output from the snippet will appear with a gray background. As example:

```
5.times: print "Hello world!"
```

```
Hello world!
Hello world!
Hello world!
Hello world!
Hello world!
```

Figure 1: HAL says: "Hello world!"

2 Features[1]

2.1 Clean syntax

`HAL` can be easily used to create **D**omain-**S**pecific **L**anguages! Parentheses in calls are optional, `self` is implicitly set accordingly to the **scope** and **blocks** can be defined easily with indentation!

```
class Array:
  def sort!:
    return self if size < 2
    p = first
    q = pop!
    lesser = q.filter with x: x < p
    greater = q.filter with x: x >= p
    lesser.sort! ++ [p] ++ greater.sort!

a = [1, 2, 3, -1, -2, -3, 20, 40, 1, 2, 200, -5]
print a.sort!
```

```
[-5, -3, -2, -1, 1, 1, 2, 2, 3, 20, 40, 200]
```

Figure 2: `HAL` can quicksort!

2.2 Everything is an object

That's why we said that `HAL` has a consistent object-oriented architecture. Even `none` is an object!

```
print \
  1.add(2), Array.new, "Chunky bacon!".size,
  &range, &range.arity, Class.repr, none.none?
```

```
3
[]
13
range
1
Class
true
```

Figure 3: Objects everywhere

[1]Discover them by yourself! Open the interactive interpreter (`bin/hal`) and play!

All the instructions return objects in `HAL`. The value returned by the last instruction of a method is the returned value of the method.

2.3 Every method can be overriden

Like in `Python` there is no method visibility concept, which means that all methods can be called from everywhere. Also, like in `Ruby` any class can be reopened at any moment to define or override methods. It is even possible to override builtin methods[2]!

```
def concat: [x for x in range(0, 10)] ++
[x for x in range(10, 20)]

print concat

class Array:
  def __concat__ x:
    x.each with index, element:
      self[index] = self[index] + element
    self

print concat
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
[10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

Figure 4: Overriding `Array` concatenation!

[2]Native methods written in Java.

2.4 Lambda blocks

Like in Ruby, it is possible to pass lambda blocks to methods. However, in HAL you can define the block using the ":" keyword and indenting accordingly. In a method, the names block_given? and yield are set depending to whether some block was given or not.

```
def list title => none, numbered? => false:
  print "<h1>" + title + "</h1>" if not title.none?
  print "<ol>" if numbered? else print "<ul>"
  yield
  print "</ol>" if numbered? else print "</ul>"

def item x:
  if block_given?:
    print "<li>" + x
    yield
    print "</li>"
  else:
    print "<li>" + x + "</li>"

list "Shopping list":
  item "Meat":
    list:
      item "Bacon"
  item "Vegetables":
    list numbered? => true:
      item "Cabbage"
      item "Cucumber"
```

```
<h1>Shopping list</h1>
<ul>
<li>Meat
<ul>
<li>Bacon</li>
</ul>
</li>
<li>Vegetables
<ol>
<li>Cabbage</li>
<li>Cucumber</li>
</ol>
</li>
</ul>
```

Figure 5: Generating HTML lists with HAL!

2.5 First-class methods

Methods are objects too! The `&` accessor can be used to avoid calling and obtain the value stored under a name.

```
def lambda: &yield

add = lambda with x, y: x + y

print add 5, add 3, add 1, 2
print &add.arity
```

```
11
2
```

Figure 6: Creating lambdas in one line of code!

2.6 Parameter groups

`HAL` method definitions support parameter grouping! The parameter marked with $*$ will store the arguments that are not captured by other parameters in an `Array`. `Arrays` can be flattened using $*$ to pass its elements as arguments!

```
def reverse head, *rest:
  [head] if rest == [] else reverse(*rest) << head

print range 20
print reverse *range 20
```

```
0...20
[19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Figure 7: Dealing with arrays with parameter grouping

2.7 Keyword parameters

`HAL` methods support keywords too! A *keyword* is a parameter that has a default value, thus it is optional to provide an argument for it. In `HAL` keywords are defined using the `=>` symbol.

```
def greet name => "John", age => 21:
  print "My name is " + name.capitalize +
" and I'm " + age.str

greet
greet "manel", 30
greet "manel", age => 30
greet age => 30, name => "manel"
```

```
My name is John and I'm 21
My name is Manel and I'm 30
My name is Manel and I'm 30
My name is Manel and I'm 30
```

Figure 8: A simple example using keywords

2.8 Module imports

Like in `Python` every file that contains code is a **module**. Modules can be imported inside other modules using the `import` statement.

```
import hal

hal.disconnect!
```

```
I'm afraid I can't do that.
```

Figure 9: Importing some examples

2.9 Four levels of scopes

`HAL` has four different variable scopes. Variables can be defined in those scopes using different access operators:

Local Without any accessor

Instance Using the "`@`" accessor

Static Using the "`@@`" accessor

Module Instance variables defined in the current module

When a name is referenced without any accessor, `HAL` searches it following that order.

```
@m = "Module variable"

class Foo:
  @a = "Class variable"

  def init:
    @a = "Instance variable"
    @m = m

  def __str__:
    @@a

print Foo.a, Foo.new.a, Foo.new.m, Foo.new

@m = "Another module variable"

print Foo.new.m
```

```
Class variable
Instance variable
Module variable
Class variable
Another module variable
```

Figure 10: Playing with the different scope levels

2.10 Inheritance

Objects can inherit methods from other objects! The name `super` contains a reference to the method with the same name implemented by the parent class, if there is any. The `init` method is called after instantiating an object.

```
class Animal:
  def init type:
    @type = type

  def __str__:
    "A " + @type

class Fox < Animal:
  def init name:
    super "fox"
    @name = name

  def __str__:
    super + " named " + @name

print Animal.new "horse"
print Fox.new "Tod"
```

```
A horse
A fox named Tod
```

Figure 11: The tale of Tod and the horse with no name

2.11 Execution of shell commands

Command shells can be executed using **backticks** (`). The value returned is the output of the execution. Additionally, it is possible to use the class `Process` to have information about the **exit status** and the **error output**.

```
File.open "test.txt" with f:
  f.write "Writing a file has never been so easy!"

print `cat test.txt`
```

```
Writing a file has never been so easy!
```

Figure 12: Writing a file and showing its contents

2.12 Native types

`HAL` defines a set of useful classes natively:

- Boolean
- Class
- Enumerable
  - Array
  - Dictionary
  - String
- File
- Kernel
  - Module
- None
- Number
  - Float
  - Integer
  - Long
  - Rational
- Object

- Package
- Process

3 Syntax

3.1 Indentation

We wanted HAL to be **readable**, **clean** and **easy to write**. We think that one of the things Python does well is the way it defines blocks by indentation because it avoids the verbosity of a *closing* token. We wanted HAL to be as non-verbose as possible. The relevant part of the grammar that allows blocks by indentation is the following:

```
}

// INDENTATION
@lexer::members
{
    public static final int MAX_INDENTS = 100;
    private int indentLevel = 0;
    private boolean end = false;
    int[] indentStack = new int[MAX_INDENTS];
    java.util.Queue<Token> tokens =
        new java.util.LinkedList<Token>();

    {
        // Compute first line indentation manually
        int i = 1;
        while(input.LA(i) == ' ') i++;

        int next = input.LA(i);

        // Ignore empty lines
        if(i > 1 && next != '\n' && next != '\r' && next != -1) {
            jump(Indent);
            indentStack[indentLevel] = i-1;
        }
    }

    @Override
    public void emit(Token t) {
        state.token = t;
        tokens.offer(t);
        //System.err.println("=> " + t);
    }

    @Override
    public Token nextToken() {
        super.nextToken();

        if(tokens.isEmpty()) {
            // End file with new line
            if(!end) {
                emit(new CommonToken(NEWLINE, "\\n"));
                end = true;
            }

            // Undo all indentation
            if(indentLevel > 0) {
                jump(Dedent);
                return nextToken();
            }
```

```
            emit(Token.EOF_TOKEN);
        }
        Token t = tokens.poll();
        //System.err.println("<= " + t);
        return t;
    }


    private void jump(int ttype) {
        String name;
        if(ttype == Indent) {
            name = "Indentation";
            indentLevel++;
        }
        else {
            name = "Dedentation";
            indentLevel--;
        }
```

It is also important, when the `NEWLINE` tokens are emited:

```
// Strings
STRING
    @init { final StringBuilder buf = new StringBuilder(); }
    : '"' ( ESC_SEQ[buf]
          | i = ~('\\'|'"') { buf.appendCodePoint(i); }
          )* '"'
      { setText(buf.toString()); }
    | '\'' ( ESC_SEQ[buf]
          | i = ~('\\'|'\'') { buf.appendCodePoint(i); }
          )* '\''
      { setText(buf.toString()); }
    ;

REGEXP
    @init { final StringBuilder buf = new StringBuilder(); }
    : 'r' DIV ( '\\' DIV { buf.append('/'); }
          | i = ~(DIV) { buf.appendCodePoint(i); }
          )* DIV
      { setText(buf.toString()); }
    ;

BACKTICKS
    @init { final StringBuilder buf = new StringBuilder(); }
    :   '`' (
            '\\`' { buf.append('`'); }
            | i = ~('`') { buf.appendCodePoint(i); }
        )* '`'
        { setText(buf.toString()); }
    ;

fragment ESC_SEQ[StringBuilder buf]
        :   '\\'
            ('b'  { buf.append('\b'); }
            |'t'  { buf.append('\t'); }
            |'n'  { buf.append('\n'); }
            |'f'  { buf.append('\f'); }
            |'r'  { buf.append('\r'); }
```

3.2 Lexical rules

The lexical rules are really simple. We tried to have a small set of keywords:

```
    :   DOUBLE_AT ID -> ^(KLASS_VAR ID)
    ;

reference_var
    :   AMPERSAND ID -> ^(REFERENCE_VAR ID)
    ;

list
    @init{boolean f = false;}
    :   LBRACK (e1=expr ((COMMA expr)* | FOR paramlist IN e2=expr {f=
        ↪ true;}))? RBRACK
        -> {f}? ^(LIST_EXPR $e2 ^(LAMBDA ^(PARAMS paramlist) ^(BLOCK
            ↪ ^(EXPR $e1))))
        -> ^(ARRAY expr*)
    ;

dict
    :
    LBRACE (entry (COMMA entry)*)? RBRACE -> ^(DICT entry*)
    ;

entry
    :   expr LARROW expr -> ^(PAIR expr expr)
    ;

access
    :   {directlyNext(LBRACK)}?
        LBRACK! expr RBRACK!
    ;

// LEXICAL RULES

// OPERATORS
ASSIGN   : '=' ;
EQUAL    : '==';
NOT_EQUAL: '!=' ;
LT       : '<' ;
LE       : '<=';
GT       : '>';
GE       : '>=';
PLUS     : '+' ;
DOUBLE_PLUS : '++';
MINUS    : '-' ;
MUL      : '*';
POW      : '**';
DIV      : '/';
DDIV     : '//';
MOD      : '%' ;
LSHIFT   : '<<';
RSHIFT   : '>>';
// KEYWORDS
NOT      : 'not';
AND      : 'and' ;
OR       : 'or' ;
TRUE     : 'true';
FALSE    : 'false';
```

```
ELIF     : 'elif';
```

```
ELSE     : 'else';
FOR      : 'for';
WHILE    : 'while';
IN       : 'in';
DEF      : 'def';
LKW      : 'with';
RETURN   : 'return';
CLASS    : 'class';
IMPORT   : 'import';
FROM     : 'from';
CASE     : 'case';
WHEN     : 'when';
RANGEI   : '..';
RANGE    : '...';
// SPECIAL SYMBOLS
COLON      : ':' ;
SEMICOLON : ';';
LPAREN   : '(';
RPAREN   : ')';
LBRACK   : '[' NL?;
RBRACK   : NL? SP? ']';
LBRACE   : '{' NL?;
RBRACE   : NL? SP? '}';
LARROW   : '=>';
LTARROW : '->';
AT       : '@';
DOUBLE_AT : '@@';
DOLLAR     : '$';
AMPERSAND : '&';
COMMA      : ',' NL?;


// Useful fragments
fragment DIGIT : ('0'..'9');
fragment LOWER : ('a'..'z');
fragment UPPER : ('A'..'Z');
fragment LETTER: (LOWER|UPPER);
fragment NL      : (('\r')? '\n')+;
fragment SP      : (' ' | '\t')+;

// Identifiers
ID   : (LETTER|'_') (LETTER|'_'|DIGIT)* (('!'|'?')('_')*)?;
SYMBOL : COLON ID;

// Numbers
NUMBER : {$type=INT;}(DIGIT+ (('.' DIGIT)=> '.' DIGIT+ {$type=FLOAT;})
    ↪ ?);
fragment INT   :;
fragment FLOAT :;

// Strings
STRING
    @init { final StringBuilder buf = new StringBuilder(); }
    : '"' ( ESC_SEQ[buf]
          | i = ~('\\'|'"') { buf.appendCodePoint(i); }
          )* '"'
      { setText(buf.toString()); }
    | '\'' ( ESC_SEQ[buf]
          | i = ~('\\'|'\'') { buf.appendCodePoint(i); }
          )* '\''
      { setText(buf.toString()); }
    ;
```

```
REGEXP
    @init { final StringBuilder buf = new StringBuilder(); }
    : 'r' DIV ( '\\' DIV { buf.append('/'); }
          | i = ~(DIV) { buf.appendCodePoint(i); }
          )* DIV
      { setText(buf.toString()); }
    ;

BACKTICKS
    @init { final StringBuilder buf = new StringBuilder(); }
    :   '`' (
            '\\`' { buf.append('`'); }
            | i = ~('`') { buf.appendCodePoint(i); }
        )* '`'
        { setText(buf.toString()); }
    ;

fragment ESC_SEQ[StringBuilder buf]
        :   '\\'
            ('b'  { buf.append('\b'); }
            |'t'  { buf.append('\t'); }
            |'n'  { buf.append('\n'); }
            |'f'  { buf.append('\f'); }
            |'r'  { buf.append('\r'); }
            |'"'␣{␣buf.append('"'); }
            |'\'' { buf.append('\''); }
            |'\\' { buf.append('\\'); } )
        ;
```

Thus giving the programmer more freedom to define its own *keywords*.

```
def unless expr:
  if not expr:
    yield

unless 1 == 0:
  print "Everything works as expected"
```

```
Everything works as expected
```

Figure 13: It's easy to create new *expressions* in `HAL`. Notice how similar to `if` looks.

3.3 Grammatical rules

The grammar is a bit complex. We engineered it thinking in HAL (the result) instead of thinking in the grammar itself. Because of that, we used some helpful Java functions, using multiple features of antlr to disambiguate some cases:

```
}
// END INDENTATION

@parser::members {
  public boolean before(int before, int type) {
    int i = 1;
    Token t;

    do {
        t = input.LT(i);
        i++;

        if(t.getType() == type)
            return true;
    } while(t.getType() != EOF && t.getType() != before);

    return false;
  }

  public boolean space(TokenStream input) {
    return !directlyFollows(input.LT(-1), input.LT(1));
  }

  private boolean directlyFollows(Token first, Token second) {
    CommonToken firstT = (CommonToken) first;
    CommonToken secondT = (CommonToken) second;

    if (firstT.getStopIndex() + 1 != secondT.getStartIndex())
      return false;

    return true;
  }

  public boolean directlyNext(int type) {
    if(input.LT(1).getType() != type)
      return false;

    return directlyFollows(input.LT(-1), input.LT(1));
  }

  public boolean nextIs(int... types) {
    int type = input.LT(1).getType();

    for(int i = 0; i < types.length; ++i) {
        if(types[i] == type)
            return true;
    }

    return false;
  }

  public boolean keywordIsNext() {
```

Some important features of the grammar are:

- Statements with if at the end (`a if cond else b`).
- List expressions (`[i for i in range 3] == [0,1,2]`).
- Statements such if, for, while or import.
- *Intelligent* spaces in arguments lists:

```
def a: 5
a - 1 \ 4
a-1   \ 4
a -1  \ Call a with argument -1
```

- Function calls with optional parentheses
- Multiline indented lambda blocks

```
// GRAMMAR

prog
    :   (stmt)* EOF -> ^(BLOCK stmt*)
    ;

stmt
    :   simple_stmt
    |   compound_stmt
    |   NEWLINE!
    ;

simple_stmt
    @init{boolean conditional = false;}
    :   s1=small_stmt (
                (options {greedy=true;}:SEMICOLON small_stmt)*
                    ↪ SEMICOLON?
            |   IF {conditional=true;} expr (ELSE s2=small_stmt)?
        )
        NEWLINE
        -> {conditional}? ^(IF_STMT expr ^(BLOCK $s1) ^(BLOCK $s2)?)
        -> small_stmt+
    ;

small_stmt
    :   assign_or_expr
    |   r=RETURN expr -> ^(RETURN[$r, "RETURN"] expr)
    ;

assign_or_expr
    :   expr (a=ASSIGN assign_or_expr)? // Right-associative
        -> {a==null}? ^(EXPR expr)
        -> ^(ASSIGN expr assign_or_expr)
    ;

compound_stmt
```

```
    :   if_stmt
    |   for_stmt
    |   while_stmt
    |   import_stmt
    |   classdef
    |   fundef
    |   do_lambda
    |   assign_lambda
    |   case_stmt
    ;

if_stmt
    :   IF if_body -> ^(IF_STMT if_body)
    ;

if_body
    :   expr COLON! block if_extension?
    ;

if_extension
    :   ELIF if_body -> ^(BLOCK ^(IF_STMT if_body))
    |   ELSE! COLON! block
    ;

for_stmt
    :  FOR paramlist IN expr COLON block
        -> ^(FOR_STMT expr ^(LAMBDA ^(PARAMS paramlist) block))
    ;

while_stmt
    :  WHILE expr COLON block -> ^(WHILE_STMT expr block)
    ;

import_stmt
    :  IMPORT module -> ^(IMPORT_STMT module)
    |  FROM module IMPORT ID (COMMA ID)* -> ^(IMPORT_STMT module ID+)
    ;

case_stmt
    :  CASE expr COLON NEWLINE
       Indent (when_stmt|NEWLINE)+
       (ELSE COLON block NEWLINE*)? Dedent -> ^(CASE_STMT expr ^(CASES
           ↪  when_stmt+) block?)
    ;

when_stmt
    :  WHEN^ expr COLON! block
    ;

module
    :  ID ('.'! ID^)*
    ;

block
    :   simple_stmt -> ^(BLOCK simple_stmt)
    |   multiline_block
    ;

multiline_block
    :   NEWLINE Indent (stmt)+ Dedent -> ^(BLOCK (stmt)+)
    ;
```

```
classdef
    :   CLASS ID ('<' expr)? COLON block -> ^(CLASSDEF ID ^(PARENT
        ↪ expr?) block)
    |   CLASS '<<' expr COLON block -> ^(EIGENCLASS expr block)
    ;

fundef
    :   DEF ID params COLON block -> ^(FUNDEF ID params block)
    ;

params
    :   paramlist? -> ^(PARAMS paramlist?)
    ;

paramlist
    :   (ID | param_group) (COMMA! (ID | param_group))* (COMMA!
        ↪ keyword)*
    |   keyword (COMMA! keyword)*
    ;

param_group
    :   '*' ID -> ^(PARAM_GROUP ID)
    ;

keyword
    :   {keywordIsNext()}?
        ID LARROW expr -> ^(KEYWORD ID expr)
    ;

funcall
    :   (options {greedy=true;}: ID args lambda_inline?) -> ^(FUNCALL
        ↪ ID args lambda_inline?)
    ;

args
    :   {directlyNext(LPAREN)}?=> LPAREN arglist? RPAREN -> ^(ARGS
        ↪ arglist?)
        | space_arglist? -> ^(ARGS space_arglist?)
    ;

space_arglist
    :   {space(input) && (!input.LT(1).getText().equals("-") ||
            directlyFollows(input.LT(1), input.LT(2)))}?
        arglist
    ;

arglist
    @init{boolean keywords=false;}
    :  (flatten_arg | expr | keyword {keywords=true;}) (options {
        ↪ greedy=true;}: COMMA! (
            {keywords==false}?=> (flatten_arg | expr | keyword {
                ↪ keywords=true;})
        |   keyword {keywords=true;}
        ))*
    ;

flatten_arg
    : {input.LT(1).getText().equals("*") && directlyFollows(input.LT
        ↪ (1), input.LT(2))}?
      '*' expr -> ^(FLATTEN_ARG expr)
    ;
```

```
do_lambda
    :   {!nextIs(FOR, WHILE, IF, ELSE, ELIF, DEF, CLASS)
        && before(NEWLINE, COLON) && !before(COLON, ASSIGN)}?
        methcalls lambda -> ^(LAMBDACALL methcalls lambda)
    ;

methcalls
    :   (atom -> atom) ('.' f=funcall -> ^(METHCALL $methcalls $f))*
    ;

assign_lambda
    :   {before(NEWLINE, COLON) && before(COLON, ASSIGN)}?
        expr ASSIGN^ do_lambda
    ;

lambda
    :
        (LKW paramlist)? COLON block -> ^(LAMBDA ^(PARAMS paramlist?)
            ↪ block)
    ;

lambda_inline
    :   {nextIs(LBRACE)}?
        LBRACE paramlist? LTARROW expr RBRACE -> ^(LAMBDA ^(PARAMS
            ↪ paramlist?) ^(BLOCK ^(EXPR expr)))
    ;

expr
    :   (boolterm -> boolterm) (options {greedy=true;}:
            (options {greedy=true;}: OR b=boolterm -> ^(OR $expr $b))+
        |   (RANGE r1=boolterm -> ^(RANGE $expr $r1))
        |   (RANGEI r2=boolterm -> ^(RANGEI $expr $r2))
        )?
    ;

boolterm
    :   boolfact (options {greedy=true;}: AND^ boolfact)*
    ;

boolfact
    :   shift_expr (options {greedy=true;}:
            (EQUAL^ | NOT_EQUAL^ | LT^ | LE^ | GT^ | GE^) shift_expr)?
    ;

shift_expr
    :   num_expr (options {greedy=true;}: (LSHIFT^ | RSHIFT^) num_expr
        ↪ )*
    ;

num_expr
    :   term (options {greedy=true;}: (PLUS^ | MINUS^ | DOUBLE_PLUS^)
        ↪ term)*
    ;

term
    :   power (options {greedy=true;}: (MUL^ | DIV^ | DDIV^ | MOD^)
        ↪ power)*
    ;

power
    :   factor (options {greedy=true;}: POW^ factor)*
    ;
```

```
factor
    :   NOT^ item
    |   MINUS item -> ^(MINUS["NEGATE"] item)
    |   item
    ;

item
    :   (atom -> atom) (options {greedy=true;}:
            a=access -> ^(GET_ITEM $item $a)
        |   ('.' f=funcall -> ^(METHCALL $item $f))
        )*
    ;

atom
    :   INT
    |   FLOAT
    |   STRING
    |   REGEXP
    |   BACKTICKS
    |   SYMBOL
    |   (b=TRUE | b=FALSE)  -> ^(BOOLEAN[$b,$b.text])
    |   NONE
    |   global_var
    |   instance_var
    |   klass_var
    |   reference_var
    |   list
    |   dict
    |   funcall // An ID can be considered a "funcall" with 0 args
    |   LPAREN! expr RPAREN!
    ;

global_var
    :   DOLLAR ID -> ^(GLOBAL_VAR ID)
    ;

instance_var
    :   AT ID -> ^(INSTANCE_VAR ID)
    ;
```

The major problem we encountered with the syntax is that a call with a lambda block is not an expression, but a call without a lambda block is (this will be fixed soon!). Thus, we needed to help the parser to **differentiate** between these two rules that start in the same way. To do so, we used the : keyword present in a call with a lambda block **before the end of the line**: iff it is present, then is a lambda block call. There are other rules that use interesting tricks (checking token separation, spacing...).

4 Interpreter

In this section we try to explain how `HAL` is structured and works internally, focusing in the most relevant parts of the language.

4.1 Methods

How does `HAL` difference between a variable access and a method call? There's no difference, they are always calls. Every `HalObject` implements the `call` method. This method has three parameters: the instance to be set as `self`, a `HalMethod` (used to pass `HalLambda`, if any) and an instance of `Arguments` which contains the arguments of the call. The `call` method needs to return a `HalObject`. Then, using polymorphism on this method, every type can act differently when it's called. Actually, this method returns the object itself by default and it's only overriden in the `HalMethod` class. Also, this polymorphism is really useful because the interpreter does not need to take into account if a `HalMethod` is a `Builtin`[3] or a method defined in `HAL` itself that needs to be interpreted, it delegates the responsability to the type.

```
abstract public class HalMethod extends HalObject<MethodDefinition>
{
    public HalMethod(MethodDefinition def) {
        super(def);
    }

    public HalObject call(HalObject instance, HalMethod lambda,
        ↪ Arguments args) {
        return mcall(instance, lambda, value.params.fill(args));
    }

    public HalObject mcall(HalObject instance, HalMethod lambda,
        ↪ Arguments args) {
        throw new RuntimeException("mcall not implemented");
    }
```

All the classes that inherit from `HalMethod` need to define the method `mcall`. The method `mcall` is exactly the same as `call` but with the arguments processed, validated and ready to be used by the method.

4.2 Builtin methods

There are two name references for every builtin provided with the interpreter: one that is surrounded with "__" (double underscores) and another that is not (`__range__` and `range` for example). The name surrounded with underscores is the one that the interpreter can use internally in case it needs it. This means that overriding this methods you can get in the way of how the interpreter does things and use it at your favour. For example, you can

[3] Natively implemented in `Java`

filter arrays on creation overriding the `__append!__` method (see Figure 14). The reference without underscores is provided for readability purposes and as a backup.

```
def ugly_array: [1, 2, [], [], 3, [[]], 4, 5, 6]

print ugly_array

class Array:
  def __append!__ x:
    append! x if x != [] else self

print ugly_array
```

```
[1, 2, [], [], 3, [[]], 4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

Figure 14: Filtering empty `Arrays` on construction

4.3 Types

It's relatively easy to add more native types to `HAL`. Normally it would only consist in writing one `Java` class that represents the type. Many types in `HAL` are a mere interface to existing types in `Java`. We thought datastructures such as Hashes, Rationals or Files could be useful, so we added them as builtins.

A type needs to:

1. Extend `HalObject`
2. Define its methods
3. Define the `getKlass` method that returns the `HalClass` of the type.

```java
public class HalFile extends HalObject<PrintWriter> {
    public HalFile(PrintWriter writer) {
        super(writer);
    }

    public HalBoolean bool() {
        return new HalBoolean(true);
    }

    private static final Reference __open__ = new Reference(new
        ↪ Builtin("open", new Params.Param("path")) {
        @Override
        public HalObject mcall(HalObject instance, HalMethod lambda,
            ↪ Arguments args) {
            try {
                HalFile file = new HalFile(new PrintWriter(args.get("
                    ↪ path").toString(), "UTF-8"));

                if(lambda != null) {
                    lambda.call(instance, null, file);
                    file.value.close();
                }

                return file;
            } catch (FileNotFoundException e) {
                throw new OSException(e.getMessage());
            } catch (UnsupportedEncodingException e) {
                throw new OSException(e.getMessage());
            }
        }
    });

    private static final Reference __print__ = new Reference(new
        ↪ Builtin("print", new Params.ParamGroup("stuff")) {
        @Override
        public HalObject mcall(HalObject instance, HalMethod lambda,
            ↪ Arguments args) {
            HalArray stuff = (HalArray) args.get("stuff");
            HalObject s = stuff.methodcall("__join__", new HalString("
                ↪ \n"));

            ((HalFile)instance).value.println(s.toString());
            return HalNone.NONE;
        }
    });

    private static final Reference __write__ = new Reference(new
        ↪ Builtin("write", new Params.ParamGroup("stuff")) {
        @Override
        public HalObject mcall(HalObject instance, HalMethod lambda,
            ↪ Arguments args) {
            HalArray stuff = (HalArray) args.get("stuff");
            HalObject s = stuff.methodcall("__join__");

            ((HalFile)instance).value.print(s.toString());
            return HalNone.NONE;
        }
    });
```

Figure 15: `File` type implementation in `Java`

5 HalTeX

As a *real life* example and to be sure that `HAL` is powerful enough at this early stage, we created a *DSL* to write in LaTeX. The output from LaTeX is very beautiful, but the syntax sometimes can be a bit cumbersome. In this section we will detail the implementation of such *DSL*.

5.1 `__method_missing__`

The most powerful thing to create *DSL*s in `HAL` (apart from the syntax) is the special method `__method_missing__`. This method can be defined in any level of the scope (instance, class, module, kernel). When in a scope a variable/function is undefined, `__method_missing__` is called in that scope. The default behaviour is to check for the variable/function in the upper level (raising a `NameException` when it's not defined in the outermost level). This can be modified, an allows the programmer to control even more the scope. One typical use of this could be *barewords*.

```
class Kernel:
    def __method_missing__ name, str => '':
        name + ' ' + str

s = This creates a string
print s
```

```
This creates a string
```

Figure 16: Simple example using `method_missing` special method.

5.2 Use example

This document was written using this *DSL* (we call it `HalTeX`). `HalTeX` comes as a builtin module and the source code can be found in the `bin/lib/` directory. The source code of this document is also included in the `doc/` directory.

```
import haltex

section 'Use example'
enumerate:
  item; p '*First item*'
  item; p '|second|'
  item; p '**third**'
```

```
\section{Use example}
\begin{enumerate}
\item
\emph{First item}
\item
\texttt{second}
\item
\textbf{third}
\end{enumerate}
```

Figure 17: This should be applicable to many markup languages.