The RPAL Functional Language

Programming Languages
Lecture 4

Adeesha Wijayasiri

Introduction to RPAL

- RPAL is a subset of PAL
- PAL: Pedagogic Algorithmic Language.
- Developed by J. Wozencraft and A. Evans at MIT, early 70's.

Why study RPAL?

- Not familiar
- Easy to study the (operational) specs.
- Good example of the operational approach for describing semantics.

Three Versions of PAL: RPAL, LPAL, and JPAL

- We will only cover RPAL.
- R in RPAL stands for right-reference (as in C).
- An RPAL program is simply an expression.
- No notion of assignment, or even memory.
- No loops, only recursion.

Two Notions: Function Definition and Function Application

- RPAL is a functional language.
- Every RPAL program is an expression.
- Running an RPAL program consists of evaluating the expression.
- The most important construct in RPAL is the function.

Two Notions: Function Definition and Function Application (cont'd)

- Functions in RPAL are first-class objects. Programmer can do anything with a function:
  - Send a function as a parameter to a function
  - Return a function from a function.

Sample RPAL Programs:

```
1) let X=3
     in
   Print(X,X**2)
   //  Prints (3,9)
```

Sample RPAL Programs:

```
2) let Abs N =
        N ls 0 -> -N | N
   in
   Print(Abs -3)
   // Prints 3
```

Preview of Lambda Calculus

- Program 1 is equivalent to:

```
(fn X. Print(X,X**2)) 3
```

- Program 2 is equivalent to:

```
let Abs = fn N. N ls 0 -> -N | N
in Print(Abs -3)
```

which is equivalent to:

```
(fn Abs. Print(Abs -3))
   (fn N. N ls 0 -> -N | N)
```

RPAL constructs

- Operators
- Function definitions
- Constant definitions
- Conditional expressions
- Function application
- Recursion

RPAL Is Dynamically Typed

- The type of an expression is determined at run - time.

- Example:
  - `let Funny = (B -> 1 | 'January')`

RPAL Has Six Data Types:

- Integer
- Truthvalue (boolean)
- String
- Tuple
- Function
- Dummy

Type Identification Functions

- All are intrinsic functions.
- Applied to a value, return true or false:
  - `Isinteger x`
  - `Istruthvalue x`
  - `Isstring x`
  - `Istuple x`
  - `Isfunction x`
  - `Isdummy x`

Other Operations

- Truthvalue operations:
  - **`or, &, not, eq, ne`**
- Integer operations:
  - **`+, -, *, /, **, eq, ne, ls, <, gr, >, le, <=, ge, >=`**
- String operations:
  - **`eq, ne, Stem S, Stern S, Conc S T`**

Examples

-
  ```
  let Name = 'Dolly'
     in Print ('Hello', Name)
  ```

-
  ```
  let Inc x = x + 1 in Print (Inc x)
  ```

-
  ```
  let Inc = fn x. x + 1
     in Print (Inc x)
  ```

-
  ```
  Print (Inc 7) where Inc x = x + 1
  ```

Nesting Definitions

- Nested scopes are as expected.

```
let X = 3
in
let Sqr X = X**2
in
Print (X, Sqr X,
       X * Sqr X,
       Sqr X ** 2)
```

Nesting Definitions (cont’d)

```
( Print (X, Sqr X,
          X * Sqr X, Sqr X ** 2)  where Sqr X =
 X**2
)
where
X = 3
```

Parentheses required ! Otherwise

```
 Sqr X = X**2 where X=3
```

Simultaneous Definitions

```
let X=3 and Y=5 in Print(X+Y)
```

- Note the **`and`** keyword: not a boolean operator (for that we have **`&`**).
- Both definitions come into scope in the Expression **`Print(X+Y)`**.
- Different from

```
let X=3 in let Y=5 in Print(X+Y)
```

Function Definitions Within One Another

- The scope of a 'within' definition is another definition, not an expression.
- Example:

```
let c=3 within f x = x + c
in Print(f 3)
```

Functions

- In RPAL, functions are first-class objects.

- Functions can be named, passed as parameters, returned from functions, selected using conditional, stored in tuples, etc.

- Treated like 'values’

Every function in RPAL has:

- A bound variable (its parameter)
- A body (an expression)
- An environment (later)

- For example:
    - `fn X. X < 0 -> -X | X`

Functions (cont'd)

- Naming a Function

```
let Abs = fn X. X ls 0 -> -X | X in Print (Abs(3))
```

- Passing a function as a parameter:

```
let f g = g 3 in let h x = x + 1 in Print(f h)
```

- Returning a function from a function:

```
let f x = fn y. x+y
 in Print (f 3 2)
```

Functions (cont’d)

- Selecting a function using conditional:

```
let B=true in
let f = B -> (fn y.y+1) | (fn y.y+2)
in Print (f 3)
```

- Storing a function in a tuple:

```
let T=((fn x.x+1),(fn x.x+2))
 in Print (T 1 3, T 2 3)
```

Functions (cont’d)

- N-ary functions are legal, using tuples:

```
let Add (x,y) = x+y
 in Print (Add (3,4) )
```

Function Application

- **`(fn x.B) A`**.

- Two orders of evaluation:
  1. PL order: evaluate A first, then B with **`x`** replaced with the value of A.
  2. Normal order, postpone evaluating A. Evaluate B with **`x`** literally replaced with A.

RPAL uses PL order.

Example: Normal order vs. PL order

```
let f x y = x
in Print(f 3 (1/0))
```

- Normal Order: output is 3.
- PL Order: division by zero error.

Recursion

- Only way to achieve repetition.
- No loops in RPAL.
- Use the **`rec`** keyword.
- Without **`rec`**, the function is not recursive.

Factorial

```
let rec Fact n =
            n eq 1 -> 1
              | n * Fact (n-1)
in
Print (Fact 3)
```

- Without `rec`, the scope of `Fact` would be the last line ONLY.

Example:

```
let rec length S =
   S eq '' -> 0
           | 1 + length (Stern S)
 in Print ( length('1,2,3'),
            length (''),
            length('abc')
           )
```

Typical layout: define functions, and print test cases.

Example:

```
let Is_perfect_Square N =
    Has_sqrt_ge (N,1)
    where
      rec Has_sqrt_ge (N,R) =
        R**2 gr N -> false
        | R**2 eq N -> true
        | Has_sqrt_ge (N,R+1)
in Print (Is_perfect_Square 4,
          Is_perfect_Square 64,
          Is_perfect_Square 3)
```

Tuples

- The only data structure in RPAL.
- Any length, any nesting depth.
- Empty tuple (length zero) is **`nil`**.

- Example:

```
let Bdate = (‘Jan’, 01, ‘2000')
in let Student =
   (‘John’,’Doe', Bdate, 19)
in Print (Student)
```

Arrays

- Tuples in general are heterogeneous.
- Array is special case of tuple: a homogeneous tuple (all elements of the same type).

- Example:

```
      let I=2
in let A=(1,I,I**2,I**3,I**4,I**5)
in Print (A)
```

Multi-Dimensional Arrays: Tuples of Tuples

```
let A=(1,2) and B=(3,4) and C=(5,6)
in let T=(A,B,C)
in Print(T)
```

- Triangular Array:

```
let A = nil aug 1
   and B=(2,3) and C=(4,5,6)
 in let T=(A,B,C)
 in Print(T)
```

Notes on Tuples

- `()` is NOT the empty tuple.
- `(3)` is NOT a singleton tuple.
- `nil` is the empty tuple.
- The singleton tuple is built using `aug`:

  ```
  nil aug 3.
  ```
- Build tuples using the comma, e.g. `(1,2,3)`

Selecting an Element From a Tuple

- Apply the tuple to an integer, as if it were a function.

- Example:

```
    let T = ( 1, (2,3), ('a', 4))
    in Print (T 2)
```

    Output: `(2,3)`

- Example:

```
         let T=('a','b',true,3)
     in Print(T 3,T 2)
```

Output: `(true, b)`

Extending Tuples

- Use **aug** (augment) operation.
- Additional element added to RIGHT side of tuple.
- NEW tuple is built.
- NOT an assignment to a tuple.
- In general, ALL objects in RPAL are IMMUTABLE.
- Example:

```
let T = (2,3) in let A = T aug 4 in
Print (A) // Output: (2,3,4)
```

Summary of Tuple Operations

- **`E1,E2,...,En`** tuple construction (tau)
- **`T aug E`** tuple extension (augmentation)
- **`Order T`** number of elements in **`T`**
- **`Null T`** true if **`T`** is **`nil`**, false otherwise

The @ Operator

- Allows infix use of a function.

- Example:

```
let Add x y = x + y
in Print (2 @Add 3 @Add 4)
```

Equivalent to:

```
let Add x y = x + y
in Print (Add (Add 2 3) 4)
```

Operator Precedence in RPAL, from lowest to highest

```
let fn
where
tau
aug
->
or
&
```

Sample RPAL Programs

- Example 1:

```
let Sum_list L =
   Partial_sum (L, Order L)
   where rec Partial_sum (L,N) =
     N eq 0 -> 0
     | L N + Partial_sum(L,N-1)
 in Print ( Sum_list (2,3,4,5) )
```

Sample RPAL Programs (cont’d)

- Example 2:

```
 let Vector_sum(A,B) =
    Partial_sum (A,B,Order A)
    where rec Partial_sum (A,B,N) =
      N eq 0 -> nil
      | ( Partial_sum(A,B,N-1)
          aug (A N + B N)
        ) // parentheses required
in Print (Vector_sum((1,2,3),(4,5,6)))
```

Error Conditions

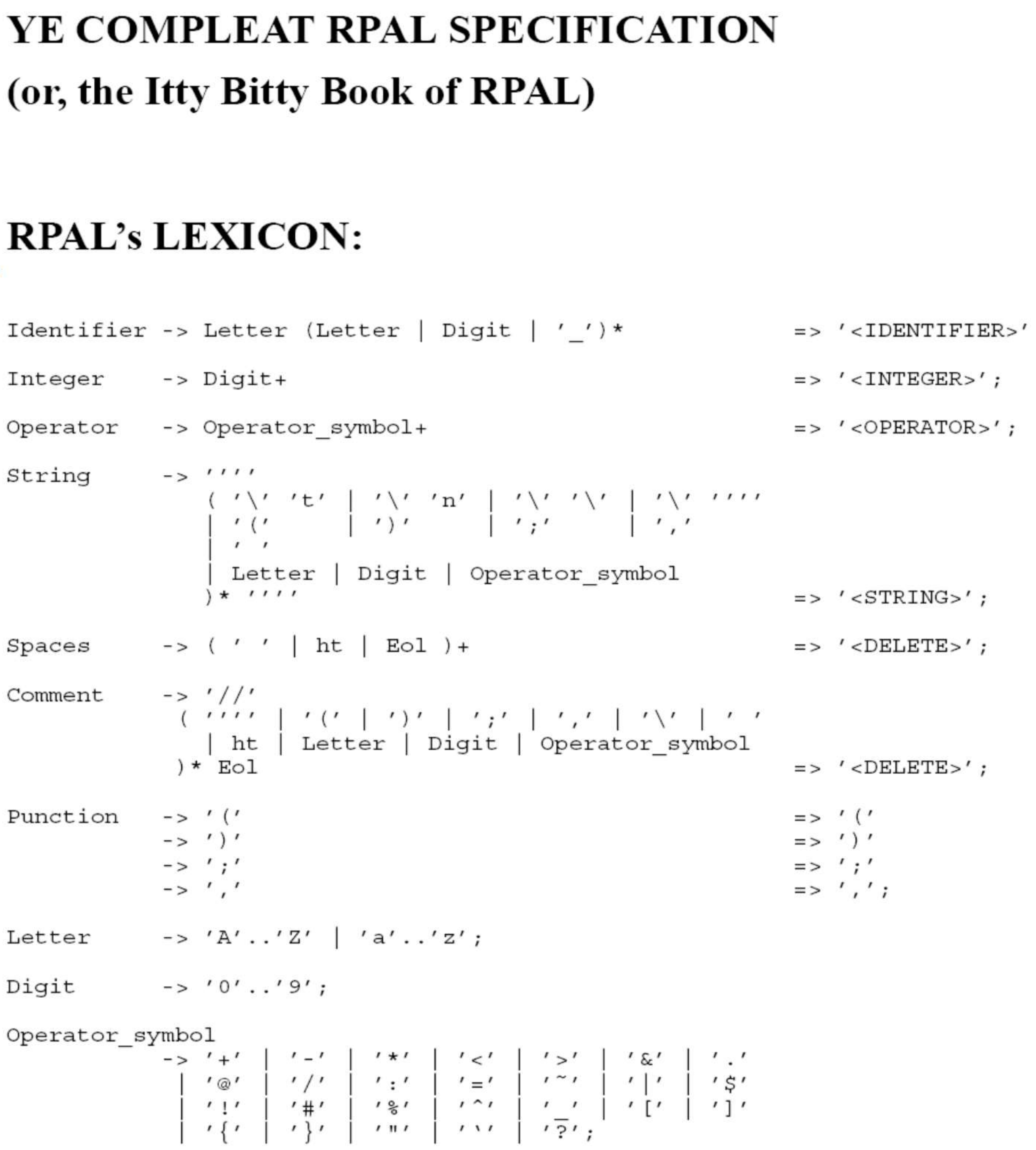
Error	Location of error
A is not a tuple	Evaluation of Order A
B is not a tuple	Indexing of B N
A shorter than B	Last part of B is ignored
B shorter than A	Indexing B N
Elements not integers	Addition

Data Verification

```
 let Vector_sum(A,B) =
     not (Istuple A) -> 'Error'
   | not (Istuple B) -> 'Error'
   | Order A ne Order B -> 'Error'
   | Partial_sum (A,B,Order A)
       where ...
in Print(Vector_sum((1,2),(4,5,6)))
```

RPAL’s SYNTAX

- RPAL’s lexical grammar.
- RPAL’s phrase-structure grammar.



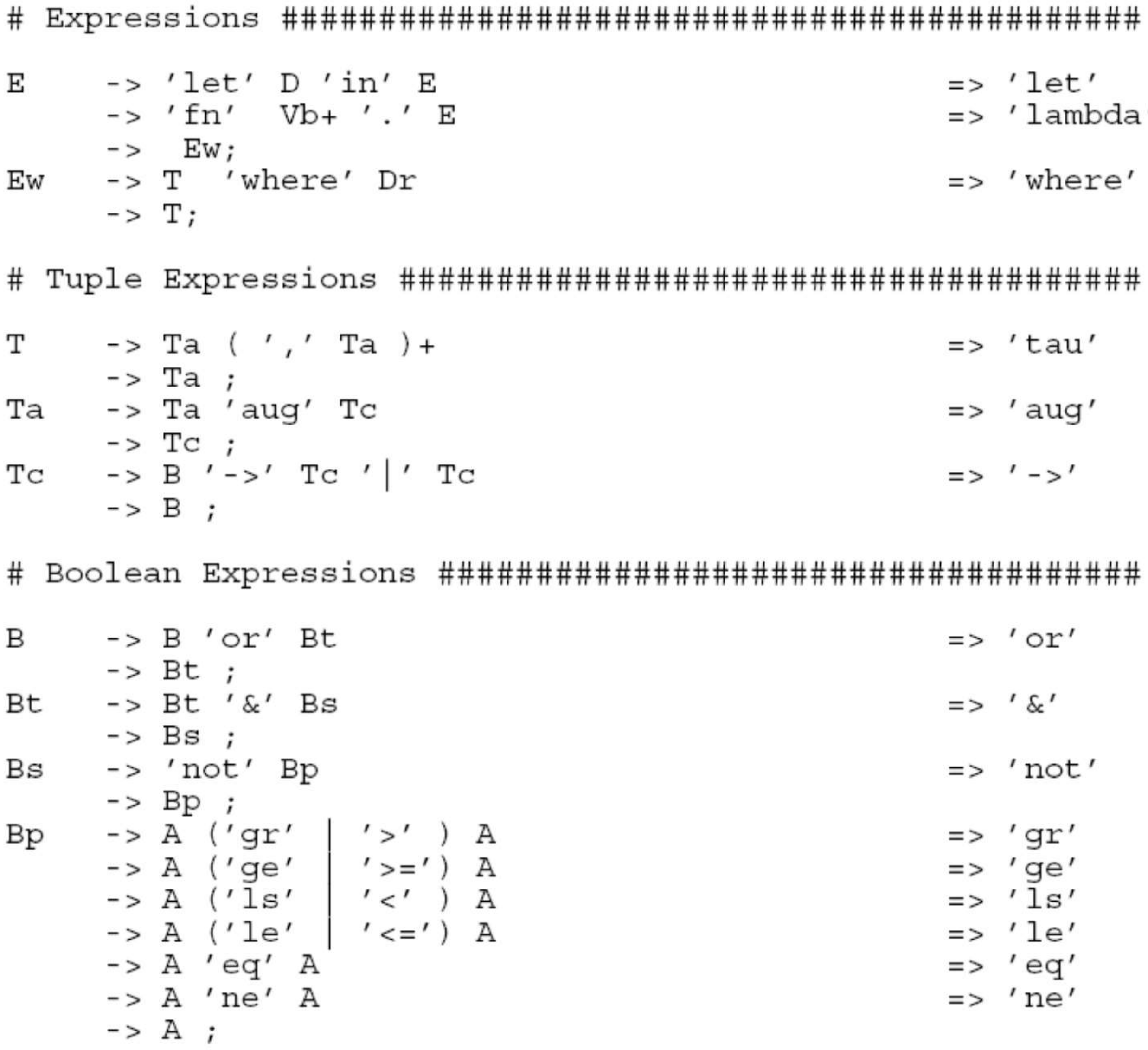
YE COMPLEAT RPAL SPECIFICATION

(or, the Itty Bitty Book of RPAL)

RPAL's LEXICON:

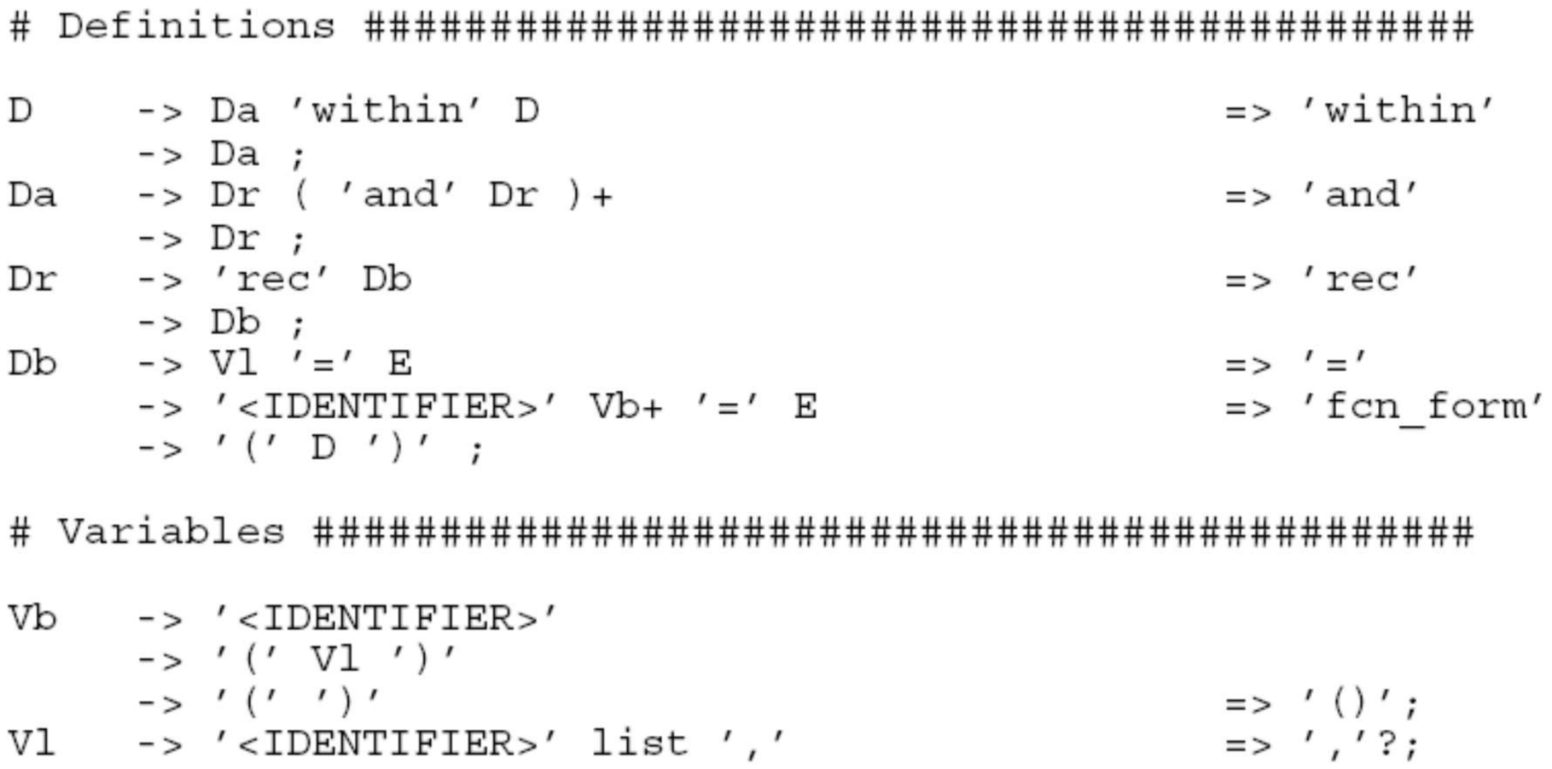
```
Identifier -> Letter (Letter | Digit | '_')*                  => '<IDENTIFIER>'

Integer    -> Digit+                                          => '<INTEGER>';

Operator   -> Operator_symbol+                                => '<OPERATOR>';

String     -> ''''
              ( '\' 't' | '\' 'n' | '\' '\' | '\' ''''
              | '('     | ')'     | ';'     | ','
              | ' '
              | Letter | Digit | Operator_symbol
              )* ''''                                         => '<STRING>';

Spaces     -> ( ' ' | ht | Eol )+                             => '<DELETE>';

Comment    -> '//'
             ( '''' | '(' | ')' | ';' | ',' | '\' | ' '
              | ht | Letter | Digit | Operator_symbol
             )* Eol                                           => '<DELETE>';

Punction   -> '('                                             => '('
           -> ')'                                             => ')'
           -> ';'                                             => ';'
           -> ','                                             => ',';

Letter     -> 'A'..'Z' | 'a'..'z';

Digit      -> '0'..'9';

Operator_symbol
           -> '+' | '-' | '*' | '<' | '>' | '&' | '.'
            | '@' | '/' | ':' | '=' | '~' | '|' | '$'
            | '!' | '#' | '%' | '^' | '_' | '[' | ']'
            | '{' | '}' | '"' | '`' | '?';
```

RPAL’s Phrase Structure Grammar:



```
# Expressions ############################################

E     -> 'let' D 'in' E                          => 'let'
      -> 'fn'  Vb+ '.' E                         => 'lambda'
      ->  Ew;
Ew    -> T  'where' Dr                           => 'where'
      -> T;

# Tuple Expressions ######################################

T     -> Ta ( ',' Ta )+                          => 'tau'
      -> Ta ;
Ta    -> Ta 'aug' Tc                             => 'aug'
      -> Tc ;
Tc    -> B '->' Tc '|' Tc                        => '->'
      -> B ;

# Boolean Expressions ####################################

B     -> B 'or' Bt                               => 'or'
      -> Bt ;
Bt    -> Bt '&' Bs                               => '&'
      -> Bs ;
Bs    -> 'not' Bp                                => 'not'
      -> Bp ;
Bp    -> A ('gr' | '>' ) A                       => 'gr'
      -> A ('ge' | '>=') A                       => 'ge'
      -> A ('ls' | '<' ) A                       => 'ls'
      -> A ('le' | '<=') A                       => 'le'
      -> A 'eq' A                                => 'eq'
      -> A 'ne' A                                => 'ne'
      -> A ;
```

```
# Arithmetic Expressions #################################

A    -> A '+' At                                => '+'
     -> A '-' At                                => '-'
     ->   '+' At
     ->   '-' At                                => 'neg'
     -> At ;
At   -> At '*' Af                               => '*'
     -> At '/' Af                               => '/'
     -> Af ;
Af   -> Ap '**' Af                              => '**'
     -> Ap ;
Ap   -> Ap '@' '<IDENTIFIER>' R                 => '@'
     -> R ;

# Rators And Rands #######################################

R    -> R Rn                                    => 'gamma'
     -> Rn ;
Rn   -> '<IDENTIFIER>'
     -> '<INTEGER>'
     -> '<STRING>'
     -> 'true'                                  => 'true'
     -> 'false'                                 => 'false'
     -> 'nil'                                   => 'nil'
     -> '(' E ')'
     -> 'dummy'                                 => 'dummy'  ;
```



```
# Definitions ############################################

D     -> Da ’within’ D                              => ’within’
      -> Da ;
Da    -> Dr ( ’and’ Dr )+                           => ’and’
      -> Dr ;
Dr    -> ’rec’ Db                                   => ’rec’
      -> Db ;
Db    -> Vl ’=’ E                                   => ’=’
      -> ’<IDENTIFIER>’ Vb+ ’=’ E                   => ’fcn_form’
      -> ’(’ D ’)’ ;

# Variables ##############################################

Vb    -> ’<IDENTIFIER>’
      -> ’(’ Vl ’)’
      -> ’(’ ’)’                                    => ’()’;
Vl    -> ’<IDENTIFIER>’ list ’,’                    => ’,’?;
```

Thank You!

REFERENCES

- Programming Language Pragmatics by Michael L. Scott. 3rd edition. Morgan Kaufmann Publishers. (April 2009).
- Slides are adopted from Lecture Slides of Dr.Malaka Walpola and Dr.Bermudez