NOTES ON RPAL

1. Introduction

RPAL is a subset of PAL, the ‘‘Pedagogic Algorithmic Language’’. There are three versions of PAL: RPAL, LPAL, and JPAL. The only one of interest here is RPAL. The ‘‘R’’ in RPAL stands for ‘‘right-reference’’, as opposed to ‘‘L’’ (LPAL) which stands for ‘‘left-reference’’. The logic behind this convention comes from a tradition in programming languages, in which an identifier, when occurring on the left side of an assignment, denotes the ‘‘left-value’’, i.e. its address, whereas if it occurs on the right side of the assignment, it denotes the ‘‘right-value’’, i.e. the value stored at that address. An RPAL program is simply an expression. RPAL has no concept of ‘‘assignment’’, nor even one of ‘‘memory’’. There are no loops, only recursion. RPAL programs, as we shall see, consist exclusively of two notions: function definition and function application. LPAL is essentially RPAL plus assignments, memory aliasing, and other issues. JPAL, finally, is LPAL plus jump statements. To repeat, we will only deal with RPAL.

RPAL is a functional language. Every RPAL program is nothing more than an expression, and ‘‘running’’ an RPAL program consists of nothing more than evaluating the expression, yielding one result. This seems superfluous at first, but the catch is that in RPAL functions are much more important than in other languages. In fact, the single most important construct in RPAL is the function. From now on, we will simply say PAL instead of RPAL. Functions in PAL are so-called ‘‘first-class’’ objects. This means that unlike traditional languages such as Pascal and C, the programmer can do anything he/she wants to with a function in PAL, including sending a function as a parameter to a function and returning a function from a function.

We first give some examples of PAL programs:

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

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

The values printed by each of these programs are (3,9) and (3), respectively. The actual value of each program is dummy.

PAL has operators (a.k.a. functors), function definitions, constant definitions, conditional expressions, function application, and recursion. The first example above illustrates a constant definition of X as 3, and function application: the function Print is applied to the pair (X,X**2). The definition of X as 3 holds in the expression Print(X,X**2); thus the result (3,9). The second example illustrates function definitions and conditional expressions: Abs is defined as a function, that takes a parameter N and returns either -N or N, depending on whether N is less than zero.

PAL is a dynamically typed language. This means that unlike strongly typed languages such as Pascal or C, they type of a variable is determined at run-time, and not earlier. For example, consider the definition

(not a complete PAL program)

let Funny = (B –> 1 | 'January')

In this definition Funny is defined as the integer 1 or the string 'January', depending on the current value of B.

PAL has six major types of values: integer, truthvalue (boolean), string, tuples, functions, and dummy. Since the language is dynamically typed, there are several functions available to find out what type of object is being dealt with. These are: Isinteger, Istruthvalue, Isstring, Istuple, Isfunction, and Isdummy. All of these are functions that take an arbitrary object, and return either true or false, depending on the type of that object.

Other features of the language:

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

2. Definitions.

Definitions in PAL are of the form ''let <defn> in <expn>''. For example:

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

Definitions can be nested arbitrarily, in which case the issue of scope appears. For example:

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

The scope of X (the value 3) is the entire expression following the first ''in''. The scope of the Sqr function is only the expression following the second ''in''.

Another form of definition is the ''where'' definition, which is similar to the ''let'' definition, except that the order is reversed. For example, the above PAL program might be re-written as:

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

Simultaneous definitions are also allowed, using the keyword ''and''. For example:

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

Note that because of this use of the keyword ''and'', the boolean (truthvalue) conjunction operator of the same name is represented using the symbol ''&''.

Finally, definitions can hold within one another, using the ‘‘within’’ clause. Normally, the scope of a ‘‘let’’ or ‘‘where’’ definition is an expression. The scope of a ‘‘within’’ definition is another definition, not an expression. For example:

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

3. Functions

In PAL, functions are first-class objects. This means that there no silly restrictions on the circumstances in which one can use a function, as there are in many other languages. In PAL, functions can be given a name using an ordinary definition, passed as parameters to other functions, returned from functions, selected from other objects in conditional expressions, and entered into data structures. Of course, functions can also be applied to actual values. Functions can be used arbitrarily in expressions, anywhere, say, an integer could be used. Every function has a so-called bound variable (its parameter) and a body (an expression). For example:

fn X. X ls 0 –> -X | X

A function can be given a name using an ordinary definition:

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

A function can be passed as parameter:

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

A function can be returned from a function:

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

A function can be selected from other objects using the conditional:

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

A function can be entered into a tuple (more on tuples later):

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

N-ary functions are allowed, by using tuples:

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

In general, a function (fn x.B) is applied to an argument A by juxtaposition, i.e. by forming the expression (fn x.B)A. There are two ways in which function application can be performed: in ‘‘PL’’ (programming language) order, and in ‘‘normal’’ order.

1) In PL order, the argument A is evaluated first, and then the body of the function (expression B) is evaluated, with x replaced by the value of A.

2) In normal order, A is not evaluated first but instead is used to replace x in B LITERALLY. Then B is evaluated.

These two are quite different, as we shall see later. For now, the following example should suffice:

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

In normal order, f is applied to 3 (with no attempt to evaluate the 3). The result is a function (fn y.3) (1/0), since the 3 replaced the x. Next, the (1/0) literally replaces all the y’s that occur in the expression 3, i.e. none. The result is 3. In PL order, the (1/0) is evaluated first, which produces an error.

4. Recursion.

Recursion is the only way to achieve repetition in PAL. Functions are not recursive by default in PAL; one must indicate that a function is recursive by using the keyword ‘‘rec’’. The classical factorial example:

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

Ordinarily (i.e. without the ‘‘rec’’ keyword) the scope of the definition of ‘‘Fact’’ would be limited to the expression on the last line. With the keyword ‘‘rec’’, however, the scope of ‘‘Fact’’ is extended to include the body of Fact itself, i.e. the expression N eq 1 –> 1 | N * Fact (N-1). Here are two examples of recursion in PAL:

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

```
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)
```

It would be instructive to simulate by hand the execution of each of these programs, and to actually run them on the computer.

5. Tuples

The only data structure available in PAL is the tuple. Tuples can be of any length (including zero for the ‘‘nil’’ tuple), and they can contain elements of any type, including other tuples. For example:

```
let Bdate = (’June’, 21, ’19XX’)
in let Me = (’Bermudez’, ’Manuel’, Bdate, 50)
in Print (Me)
```

Tuples can be used for any purpose, since they provide more flexibility than any other data structure. An array is a special case of a tuple, in which all the components are of the same type. For example:

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

A two-dimensional array can be had by constructing a tuple of tuples:

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

Even a triangular array can be had:

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

To select an element from a tuple, one ‘‘applies’’ the tuple to an integer, as if the tuple were a function. For example:

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

Tuples can also be extended, i.e. an element (of any type) can be added to the end of an existing tuple, using the ‘‘aug’’ (augment) operation. For example:

let T = (2,3) in let A = T aug 4 in Print (A)

Summarizing, here are the tuple operations:

$E_1, E_2, \ldots, E_n$	tuple construction (tau)
T aug E	tuple extension (augmentation)
Order T	number of elements of a tuple
Null T	true if T is nil, false otherwise

6. Operator Precedence in PAL

Pal is a language rich in operators. Here they are, from least to most binding, i.e from lowest to highest precedence.

let fn
where
tau
aug
–>
or
&
not
gr ge le ls eq ne
+ -
* /
**
@ <IDENTIFIER>
function application

Only the ‘‘@’’ operator remains. This is the ‘‘infix’’ operator, which allows infix use of a function. For example:

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

7. Examples

Here are two sample PAL 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) )
```

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) )
in Print ( Vector_sum ( (1,2,3), (4,5,6) ) )
```

There are many error conditions that can occur in this last program. Here is a (non)-exhaustive list:

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
Components not integers	Addition
A and B components not of same type	Addition

Here an example of how to implement one data verification:

```
let Vector_sum(A,B) =
        not (Istuple A) –> 'Error'
        | Partial_sum (A,B,Order A) where ...
```

Once again, it would be instructive to hand-simulate the execution of each of these, and to run them on the computer as well.