el — A little language for PLC

http://www.cs.rit.edu/~ats/projects/el/doc/

Axel Schreiner
ats@cs.rit.edu

LfL March 22, 2006
LP Course Project

- design a little language
- implement an interpreter
- add domain-specific features
- add aggregates
- add functions

el

- expression language
- lexically scoped, dynamically typed
- integer, floating point, string, and procedure values, array and hash syntax for hash tables
- six ways to pass parameters
- three interpreters with different memory models
- objects and XML as intermediate formats
Execution

- executable archive, Shell-based front-end, and (spoofed) applet

- behavior controlled by a combination of program tree visitors and Java properties

- by default interpretation with call by value as default parameter passing style

- last value is printed

Types

- Integers, in decimal, octal, or hex; arithmetic and bit operations, comparisons; widened if combined with floating point.

- Floating point in decimal, arithmetic and comparison only.

- Strings with C-escapes; concatenation with +; comparison with string; conversion with signs.
Operators

- Logic on numbers: || && !
- Comparison: <= < >= > == !=
- Contains: ~ !~ string and pattern, table and element value, key and table
- Type: is array int float proc string
- Arithmetic, Bit, and Concatenation
Control Structures

```plaintext
# iterative greatest common divisor
var x = 36; y = 54 in begin
  while x != y do
    if x > y then
      x = x - y
    else
      y = y - x;
  x # they might have been equal initially
end
```
Strings, Aggregates

```javascript
# word frequencies
var table = [] in
try
  while 1 do
    var words = split(getline(), "[ \t"]) in
    for i in words do
      if words[i] == "" then 0
      else if words[i] ~ table then
        table[words[i]] = table[words[i]] + 1
      else table[words[i]] = 1
    catch eof in
  table
```

Functions

# mutually recursive procedures
proc
even (x)
  if x then odd(x-1) else 1;
odd (x)
  if x then even(x-1) else 0;
in
odd(13)
try
  while 1 do
    println getline()
  catch eof in ""

length("hello");
length([1, 2, 3])

try begin
  remove([1], 0);
  remove({'one': 1}, 'one')
end catch notFound in ""

split("hello, world", " ")[1]

substr("hello", 1, 3)
Programming Language Concepts

Parameter Passing

http://www.cs.rit.edu/~ats/projects/el/doc/

Terminology

\[ \text{var } i = 10 \]
\[ \text{in} \]
\[ \text{proc } f \ (x) \]
\[ \quad \begin{align*}
\text{begin println } & \ i + \ " \ " + x; \ " \end{align*} \]
\[ \text{end} \]
\[ \text{in} \]
\[ f(i+1) \]

- **Parameter:** \( x \)
- **Operand:** \( i+1 \)
- **Argument:** value of operand
by value

- Argument (operand value) is bound to parameter.

- C, Java, Scheme procedures, \textit{el in}
by reference

```plaintext
var i = 10
in
proc f (x)
  println x = 11
in
  begin f(i); i end
```

- If operand is assignable, the parameter is an alias for it — assignment to parameter changes operand variable.
- Fortran, `el ref`
by value return

```plaintext
var i = 10
in
proc f (x)
    begin x = 11; println x+" "+i end
in
    begin f(i); i end
```

- By value, and if operand is assignable the final value of the parameter is assigned to it.
- Algol W, CORBA IDL, el inout
by return value

```plaintext
var i = 10
in
  proc f (x)
  begin x = x+2*i; println x+" "+i end
in
  begin f(i); i end

Parameter is not initialized; if operand is assignable, the final value of the parameter is assigned to it.

CORBA IDL, el out
```
by name

```plaintext
var i = 0
in
proc f (n, two_n)
    begin n = 5; two_n end
in
f(i, 2*i)
```

Operand is bound to parameter and evaluated in it's original context whenever the parameter is referenced.

Algol, Scheme macros, el name
by name

proc repeat (body, until)
    begin body; while !until do body end
in
    var i = 1
    in repeat(print i+" ", (i = i+1) > 3)

proc jensen (i, a_i)
    while i >= 0 do
        begin print a_i; i = i-1 end
in
    var i = 2; a = [ 1, 2, 3 ]
in begin jensen(i, a[i]); "" end
by need

var i = 0
in
    proc f (n, two_n)
    begin n = 5; two_n; n = 6; two_n end
in
    begin println f(i, 2*i); i end

- By name, but first operand value is memoized.
- Haskell, el need
Visitors

el.Hash: interprets using nested hashtables mapping names to denoted values
el.call

el.Alloc: annotates with static or nesting depth and offset in activation record
el.nest

el.Heap: interprets using static link and garbage-collected activation records
el.call

el.Stack: interprets using display and stacked activation records (restricts procs)
el.call

el.Dump: dumps as indented text (cannot be read again)
el.call

el.Xml: dumps as XML (read with el.Xel)
el.dtd

el.lineno
## Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Default Parameter Passing Style</th>
<th>Count Contours (out) or Nesting Depth (in)</th>
<th>Add DTD Reference</th>
<th>Store Source Line Number in XML</th>
<th>Validate (Requires DTD Reference)</th>
</tr>
</thead>
<tbody>
<tr>
<td>el.call</td>
<td><code>in</code>, <code>inout</code>, <code>name</code>, <code>need</code>, <code>out</code>, <code>ref</code></td>
<td>interpreters</td>
<td>default parameter passing style</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>el.nest</td>
<td><code>true</code>, <code>false</code></td>
<td>el.Alloc</td>
<td>count contours (out) or nesting depth (in)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xel.dtd</td>
<td>URL</td>
<td>el.Xml</td>
<td>add DTD reference</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xel.linen0</td>
<td><code>true</code>, <code>false</code></td>
<td>el.Xml</td>
<td>store source line number in XML</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xel.validate</td>
<td><code>true</code>, <code>false</code></td>
<td>el.Xel</td>
<td>validate (requires DTD reference)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dump

$ el -tree
1+2
end of file
1       Main
1       Add
1       Integer 1
1       Integer 2
$ el -tree
1+2
d: offle
1       Main
1         Add
1           Integer 1
1           Integer 2

$ el -xml
1+2
end of file
<!DOCTYPE main SYSTEM 'el.dtd'>
<main>
  <add>
    <integer value='1'/>
    <integer value='2'/>
  </add>
</main>

$ el -tree
1+2
end of file
<!DOCTYPE main SYSTEM 'el.dtd'>
<main>
  <add>
    <integer value='1'/>
    <integer value='2'/>
  </add>
</main>
Memory Allocation

```
$ echo 'var x=1 in proc y(z) x+z in y(x)' | el -tree
1   Main
1     Let
1       Var
1         Symbol x
1         Integer 1
1     Letrec
1       Proc
1         Symbol y
1         Parm
1           Symbol z
1     Add
1       Deref
1         Symbol x
1       Deref
1           Symbol z
1     Call
1       Deref
1         Symbol y
1     Arg
1         Symbol x
```
Memory Allocation

```bash
$ echo 'var x=1 in proc y(z) x+z in y(x)' | el -alloc
1       Main 1:9
1         Let
1           Var
1             Symbol x 0:7
1             Integer 1
1           Letrec
1             Proc 2:2
1               Symbol y 0:8
1               Parm
1                 Symbol z 0:1
1               Add
1                 Deref
1                   Symbol x 1:7
1                 Deref
1                   Symbol z 0:1
1             Call
1               Deref
1                 Symbol y 0:8
1               Arg
1                 Symbol x 0:7
```
Memory Allocation

$ echo 'var x=1 in proc y(z) x+z in y(x)' | el -nest
1 Main 1:10
  1 Let
  1 Var
  1 Symbol x 1:8
  1 Integer 1
  1 Letrec
  1 Proc 2:4
  1 Symbol y 1:9
  1 Parm
  1 Symbol z 2:3
  1 Add
  1 Deref
  1 Symbol x 1:8
  1 Deref
  1 Symbol z 2:3
  1 Call
  1 Deref
  1 Symbol y 1:9
  1 Arg
  1 Symbol x 1:8
AR requires GC

\[
\text{var } a = 1; \ x = 0 \text{ in begin}
\]

\[
\text{proc } f (y)
\]

\[
\text{var } b = 2 \text{ in begin}
\]

\[
y = \text{proc } g (z)
\]

\[
\text{var } c = 3 \text{ in } a+b+c+z
\]

\[
in
g;
\]

\[
\text{print } y(4)+\ " \ "
\]

\[
\text{end}
\]

\[
in
\]

\[
f(\text{ref } x);
\]

\[
x(5)
\]

\[
\text{end}
\]
# Closure vs. AR Stack

<table>
<thead>
<tr>
<th>Algol</th>
<th>Pascal</th>
<th>C</th>
<th>Java</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested function definitions with lexical scoping</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function pointers returned</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Garbage collection</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Closure vs. AR Stack

<table>
<thead>
<tr>
<th></th>
<th>Algol</th>
<th>Pascal</th>
<th>C</th>
<th>Java</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested function definitions with lexical scoping</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Function pointers returned</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Garbage collection</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Summary

- *el* is useful for discussing...
  - a language based only on expressions
  - mixed-mode arithmetic
  - programming with hash tables
  - parameter passing styles
  - closures and their costs
  - implementation principles