l3lang(n)

Michael Hohn, mhhohn@users.sf.net

April 9, 2008

section

1 NAME

l3lang - the l3 data handling and programming language

2 SYNOPSIS

The l3 language is an interpreted, lexically scoped language designed to automate much of the parameter and data handling of the type often encountered in experimental (scientific) computing. To accomplish this, l3 programs and all their data are persistent, and can be inspected at any time; further, the language maps one-to-one onto its graphical interface to make data traversal and inspection practical. See l3gui(1) for the interface description.

The l3 language is implemented as an interpreter written in Python; all Python modules can be directly imported and used from l3. Thus, algorithms without interesting intermediate data can be written in Python and called from l3 to provide a data interface for applications, and existing Python code can be used directly.

The l3 syntax follows Python's when possible, with only a few syntactic extensions for parameter, module, and documentation handling. As result, most l3 scripts are valid Python scripts and can be moved to Python if desired.

Only a subset of Python is repeated in l3; in particular, there are no classes.

The semantics of l3 are closer to a purely functional language. Everything in l3 is an expression, including computed values. l3 is designed as a single-assignment language to permit reliable data tracking; overwriting of existing names is **strongly** discouraged (but allowed for compatibility with existing Python scripts).

l3 introduces several new language features to support direct access to expressions and their values from a random-access interface (see l3gui(1)); they are

- Every expression is turned into an object by assignment of a permanent, unique id. Both the expression and the value(s) computed by the expression can be accessed via this id (using the gui, through the script's text).
- The data flow graph of l3 scripts is a stacked dag, and can be generated from a script. In this view, named data are emphasized, and data interdependencies shown. Loops cause no "explosion" of the data flow graph.

- l3 scripts are lexically scoped, allowing for arbitrarily nested (static) namespaces and avoiding name collisions. Also, every function call introduces a new (dynamic) scope, so multiple calls to the same function do not produce naming conflicts, even when file names are used.
- All scripts and their computed data are persistent. l3 is most useful when most data is should be kept. By keeping *all* data, file I/O need not be part of a script, keeping it simpler and cleaner. Exporting data to other programs *after* script execution is simple and can be done interactively or by adding I/O commands where desired.
- Evaluation of scripts is incremental (need-based). An expression is only executed if it is out-of-date or newly inserted. This allows l3 scripts to *evolve*; they can be added to without re-running existing code, and references to all previouly computed data are available.
- Values and scripts are the same. An l3 value is a valid l3 script. Therefore, computed values from prior scripts can be directly inserted into new scripts.

3 ENVIRONMENTS

Names are bound in environments. Environments are lexically nested and dynamically formed on function entry. Scoping is lexical.

4 SEMANTICS

The L3 semantics deal with the special cases of persistence and need-based evaluation to make the language convenient to **use**, at the cost of complicating the language.

When a new L3 program is executed the special cases do not affect evaluation, so a simple description of those semantics are given first, in **??**.

Persistence and need-based evaluation introduce retention and retrieval of computed values, and a mechanism based on timestamps to decide whether to interpret to get a new value or retrieve a previously computed value. The additional rules introduced by evaluation are covered in **??**.

Other special cases are listed after.

General rules Numbers and strings evaluate to their internal representation and return that. Nested expressions evaluate children, then self, and return this value.

Lists evaluate from first to last element, and a list is returned. Programs evaluate from first to last entry, and the last computed value is returned.

Assignments are made in the enclosing environment, typically the enclosing Program, Map, or Function.

Name (Symbol) lookup is lexical, starting in the current environment and searching enclosing defining environments until a binding is found, if any.

A Map (dictionary) is a Program inside a nested environment; the Program is evaluated with bindings made in this environment, and the environment itself is returned.

Timestamps and IDs Every expression has an associated numeric id and time stamp. The id is assigned once and never changes; every expression is uniquely identified via its id. There are three logical ages an expression may have: SETUP, EXTERN, and INTEGER. Corresponding to these are the actual stamps "SETUP", "EXTERN" and any positive integer. The ordering of these is SETUP < EXTERN < INTEGER.

During evaluation, timestamps increase monotonically; they are independent of system time. An expression with time SETUP is evaluated and its time updated to the current time (a positive integer).

An expression with time EXTERN is evaluated and no time stamp change is made.

A terminal expression with an integer time stamp is not evaluated at all; its prior value is retrieved and returned. The timestamp remains unchanged.

A nested expression `E` with an intger time stamp first evaluates its children. If any child's value is newer than `E`, it is evaluated again using the new values, and `E`'s timestamp updated.

An expression of EXTERN age is assumed constant, and its time stamp never changes.

Special forms Unlike other expressions in L3, `inline` is always evaluated, so its contents must be constant. For example

External values Most application-specific values are not recognized by L3 at all. They are treated as simple values by l3 and represented as text, using their Python `str()` form (which in turn is provided by the library implementing the value).

5 PYTHON COMPATIBILITY

The following Python(1) constructs are not available in l3. The workarounds are straightforward and are valid in Python and l3.

Ungrouped tuple assignment Instead of

```
    psi, sx, sy, scale = compose_()
```

use

```
   (psi, sx, sy, scale) = compose_()
```

list assignments Instead of

```
    [psin, sxn, syn, mn] = combine_params2( ...)
```

use

```
    (psin, sxn, syn, mn) = combine_params2( ...)
```

dictionary syntax The `{ key : val }` syntax is not available. Instead of

```
    {"negative":0, "mask":mask}
```

use

```
    dict(negative = 0, mask = mask)
```

No inline blocks The shortcuts `if 1: do_this` and `def f(a,b): return a+b` are not available. Use the long forms

```
    if 1:
        do_this
```

and

```
    def f(a,b):
        return a+b
```

instead.

Values returned to l3 must pickle properly. There are several Python constructs that cannot be pickled and must not be used in l3 code.

Most generators will not pickle. In particular avoid `xrange` and use `range` instead.

Avoid using `from foo import *` in inline code. Importing this way is often a problem for pickling. For example, using

```
from numpy import *
```

causes the error

```
pickle.PicklingError: Can't pickle <type 'frame'>: it's not
found as __builtin__.frame
```

and the session state will no longer save.

6 SEE ALSO

l3(1), l3gui(1)

7 AUTHOR

Michael Hohn, mhhohn@users.sf.net

8 COPYING