pLisp: A Friendly Lisp IDE for Beginners

Rajesh Jayaprakash
TCS Research, India
rajesh.jayaprakash@tcs.com

ABSTRACT

This abstract describes the design and implementation of pLisp, a Lisp dialect and integrated development environment modeled on Smalltalk that targets beginners.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**;

KEYWORDS

lisp, integrated development environment

1 INTRODUCTION

pLisp is an integrated development environment (IDE) and an underlying Lisp dialect (based on Common Lisp) that is targeted towards beginners. It is an attempt at developing a Lisp IDE that matches (or at least approaches) the simplicity and elegance of typical Smalltalk environments and thereby hopefully providing a friendlier environment for beginners to learn Lisp.

Smalltalk environments are characterized by three interface components: the workspace, the transcript, and the system browser. The workspace and the transcript windows together serve the purpose of the canonical Read-Eval-Print Loop (REPL) used to interact with programming systems in the command-line mode, while the system browser is used to view the universe of objects available to the user and to define new objects. pLisp adopts the same idioms to model this interaction. Figures 1 and 2 illustrate sample screenshots where the user has entered an expression and has issued the command for evaluating the expression.

pLisp supports the following features:

- Graphical IDE with context-sensitive help, syntax coloring, autocomplete, and auto-indentation
- Native compiler
- Continuations
- Exception handling
- Foreign function interface
- Serialization at both system- and object level
- Package/Namespace system

The productivity-enhancing features like expression evaluation, autocompletion and auto-indentation of code, and context-sensitive help are available in all code-editing contexts (Workspace, code panels in the System Browser and Callers window, and the File Browser). Another useful Smalltalk-inspired feature implemented in pLisp is the ability to store the entire programming session—including the GUI state—in the serialized image; this enables the user to carry over the programming experience seamlessly across sessions, even in the middle of a debugging exercise.

pLisp has been released under the GPL 3.0 license and is freely available for download [1]. At present, pLisp is available for Linux (both 32-bit and 64-bit), Windows (32-bit), and Mac OS X platforms. pLisp is written in C, and relies on open-source components (GTK+,

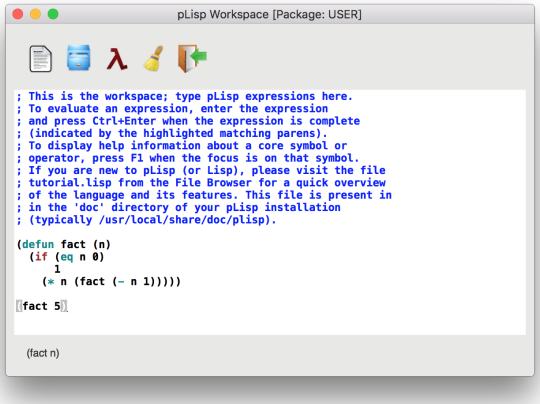

Figure 1: The pLisp Workspace window

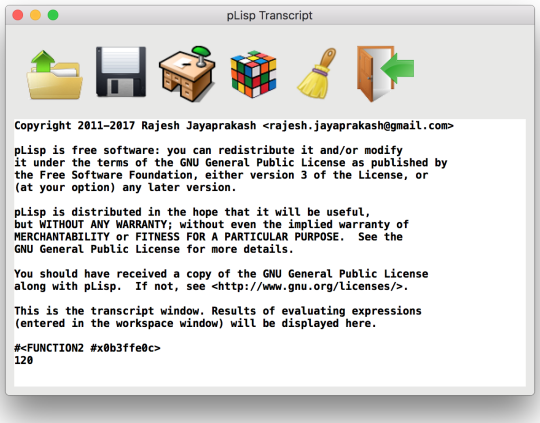

Figure 2: The pLisp Transcript window

ELS'18, April 16–17 2018, Marbella, Spain

ACM ISBN 978-2-9557474-2-1.

GtkSourceView, Tiny C Compiler, the Boehm Garbage Collector, and libffi).

2 IMPLEMENTATION

pLisp is a Lisp-1 dialect, i.e., functions share the same namespace as the other objects in the system. The syntax of pLisp closely mirrors that of Common Lisp (e.g., `defun`, `defmacro`, `progn`, and macro-related constructs like backquote, comma, and comma-at), however, notations from Scheme are also used (`call/cc`). The design philosophy of pLisp is to be more-or-less source-code compatible with Common Lisp so that users can easily transition to Common Lisp and carry over their knowledge and code.

2.1 Syntax

The pLisp s-expression grammar is shown in Figure 3. Except for the language constructs and primitive operators, the core of pLisp is written in itself. The support for continuations and the `call/cc` construct, coupled with the use of macros, enables this and the implementation of sophisticated programming constructs like loops and exception handling at the library level.

$$
\begin{aligned}
E \;::=\; & L \mid I \\
& \mid \texttt{(define } I_{name}\ E_{defn}\texttt{)} \\
& \mid \texttt{(set } I_{name}\ E_{defn}\texttt{)} \\
& \mid \texttt{(lambda (}I^{*}_{formal}\texttt{)}\ E^{*}_{body}\texttt{)} \\
& \mid \texttt{(macro (}I^{*}_{formal}\texttt{)}\ E^{*}_{body}\texttt{)} \\
& \mid \texttt{(error } E\texttt{)} \\
& \mid \texttt{(if } E_{test}\ E_{then}\ E_{else}\texttt{)} \\
& \mid \texttt{(}E_{rator}\ E^{*}_{rand}\texttt{)} \\
& \mid \texttt{(let ((}I_{name}\ E_{defn}\texttt{)}^{*}\texttt{)}\ E^{*}_{body}\texttt{)} \\
& \mid \texttt{(letrec ((}I_{name}\ E_{defn}\texttt{)}^{*}\texttt{)}\ E^{*}_{body}\texttt{)} \\
& \mid \texttt{(call/cc } E\texttt{)}
\end{aligned}
$$

Figure 3: pLisp informal s-expression grammar

2.2 Object Model

pLisp supports the following object types:

- Integers
- Floating point numbers
- Characters
- Strings
- Symbols
- Arrays
- CONS cells
- Closures
- Macros

All objects are internally represented by `OBJECT_PTR`, a `typedef` for `uintptr_t`, the C language data type used for storing pointer values of the implementation platform. The four least significant bits of the value are used to tag the object type (e.g., `0001` for symbol objects, `0010` for string literals, and so on), while the remaining *(n-4)* bits (where *n* is the total number of bits) of the value take on different meanings depending on the object type, i.e., whether the object is a boxed object or an immediate object. If the object is a boxed object, the remaining bits store the referenced memory location. The loss of the four least significant bits is obviated by making use of the `GC_posix_memalign()` call for the memory allocation and thus ensuring that the four least significant bits of the returned address are zeros.

2.3 Compiler

The pLisp compiler transforms the code to continuation-passing style (CPS) [2] and emits C code, which is then passed to the Tiny C Compiler (TCC) to produce native code. The compiler does the transformation in the following passes [3]:

- Desugaring/Macro expansion
- Assignment conversion
- Translation
- Renaming
- CPS conversion
- Closure conversion
- Lift transformation
- Conversion to C

These passes produce progressively simpler pLisp dialects, culminating in a version with semantics close enough to C. Since TCC is utilized for the native code generation, the transformation pipeline does not include passes like register allocation/spilling.

2.4 Debugger

Since pLisp uses the continuation-passing style, all the functions invoked in the course of evaluating the expression are extant at any point in time, and are displayed in the debug call stack. At present, only the break/resume functionality (and inspection of function arguments) is supported in pLisp.

The compilation process introduces a large number of internal continuation functions as part of the CPS conversion pass; the debugging infrastructure needs to filter out these continuations so that the user is presented with only those functions they need to be aware of (i.e., those that have external source representations). This is accomplished by logic in the C conversion phase, which generates code to store a closure in the debug stack only if that closure maps to a top-level definition.

3 CONCLUSION

This abstract describes the design and implementation of pLisp, a Lisp dialect and integrated development environment modeled on Smalltalk that targets Lisp beginners. While pLisp is oriented towards beginners, its feature-set is complete enough (and its performance robust enough) to serve the needs of a typical medium-sized Lisp development project. Introduction of multithreading capabilities and enhancements to the debugger to enable continuing or restarting a computation with user-supplied values are part of the future work being considered.

REFERENCES

[1] R. Jayaprakash. pLisp IDE. https://github.com/shikantaza/pLisp, 2018.

[2] Guy L Steele Jr. Rabbit: A compiler for scheme. Technical report, Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1978.

[3] Franklyn Turbak, David Gifford, and Mark A Sheldon. *Design concepts in programming languages*. MIT press, 2008.