pLisp: A Friendly Lisp IDE for Beginners

Rajesh Jayaprakash
Tata Consultancy Services
Chennai, India

Overview

- Basics
  - What is pLisp?
  - Motivation
  - Features
- Internals
  - Language
  - Object Model
  - Compiler/Debugger
  - Core library, Serialization, FFI
- Future work
- Demo
- Q&A

What is pLisp?

- A Lisp dialect based on Common Lisp
- An integrated development environment
- Platforms
  - Linux, Windows, OS X



- Open source; GPL 3.0 license
- Built using OSS components
  - GTK+, GTKSourceView, libffi, Boehm GC, TCC, Flex, Bison

All trademarks are the properties of their respective owners

Motivation

- To serve as a friendly environment for beginners to learn Lisp
  - Graduate to Common Lisp and its implementations/environments
- Inspired by Smalltalk environments
  - Workspace/Transcript/System Browser
  - Ability to edit code in all contexts
  - Image based development
    - GUI state part of image

pLisp 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

```
(defun fact (n)
  (if (eq n 0)
      1
    (* n (fact (- n 1)))))

(fact 5)
```


Beginner-Friendly Features (some inspired by Smalltalk)

- Workspace-Transcript as REPL
- System Browser
  - Useful for navigating between multiple, small functions
- Image-based development (incl. GUI state)
- Flexibility w.r.t. image-based or file-based development
- Ability to evaluate code in all contexts
  - Workspace, Browser code panel, Callers Window, File Browser
  - 'Live' tutorial

pLisp Internals - Language

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

Note on Array Syntax

- Support for a more natural array syntax

```
(define a (array (5) 0)   => [0 0 0 0 0]
(print a[0])              => 0
(define ma (array (2 2) 0)) => [[0 0] [0 0]]
(print ma[0 0])           => 0
(aset a[0] “Hello”)       => “Hello”
(print a)                 => [“Hello” 0 0 0 0]
(aset ma[0 1] 3.14)       => 3.14
(print ma)                => [[3.14 0] [0 0]]
```

- Realized through parser tweaks and macros

pLisp Internals – Object Model

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

Objects represented by **`OBJECT_PTR`**, a `typedef` for `uintptr_t`

0001 for symbols, 0010 for string literals, etc.

4-bit tag

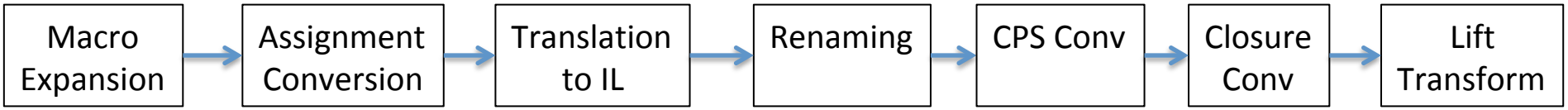
(n-4) bit value

Object-specific

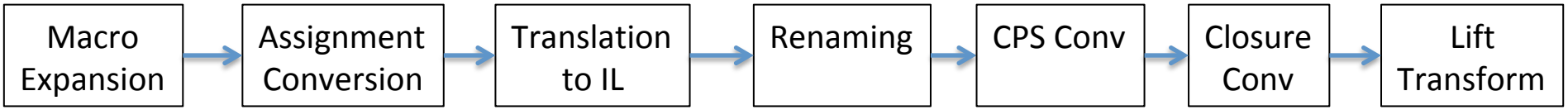
pLisp Object Model (cont.)

Object Type	Object-Specific Value
Integer	Address of allocated integer
Float	Address of allocated floating point number
Character	Numeric representation of ASCII value (e.g. 65 for ‘A’)
String	Mutable strings are arrays (see below); for immutable strings, value is an index into a global strings array
Symbol	Value is split into a) an index into a global packages array and b) an index into the strings array of the chosen packages array element
Array	Address of segment of size n+1, first element storing the integer object denoting the array size n
CONS cell	Address of first of two contiguous memory locations
Closure	Address of linked list of CONS cells containing the native function object and the closed-over objects
Macro	Similar to above
Native function	Address of native function pointer

Compiler

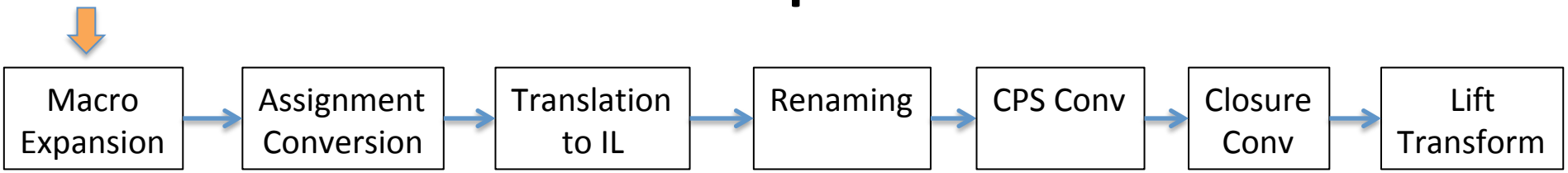

Macro Expansion
Assignment Conversion
Translation to IL
Renaming
CPS Conv
Closure Conv
Lift Transform

Compiler



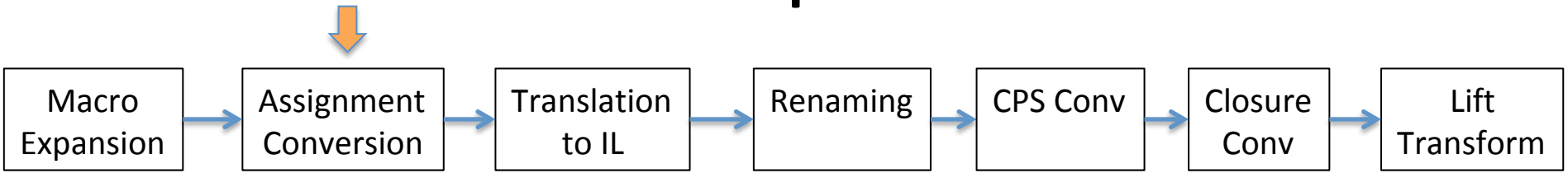

```
(print "Hello World!")
```


Compiler




```
(print "Hello World!")
```

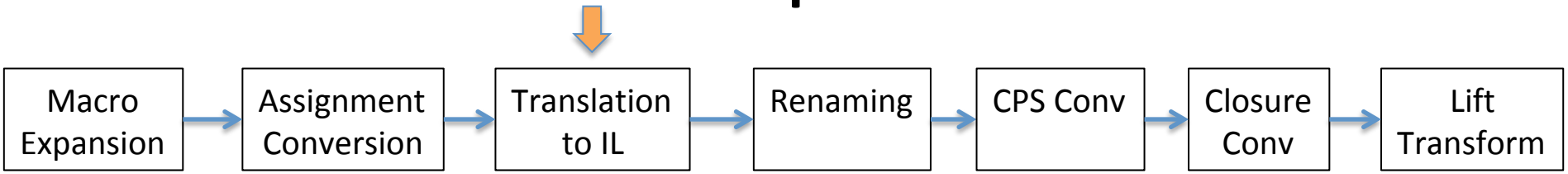

Compiler



Conversion of mutable variables into mutable cells

```
((prim-car print) "Hello World!")
```

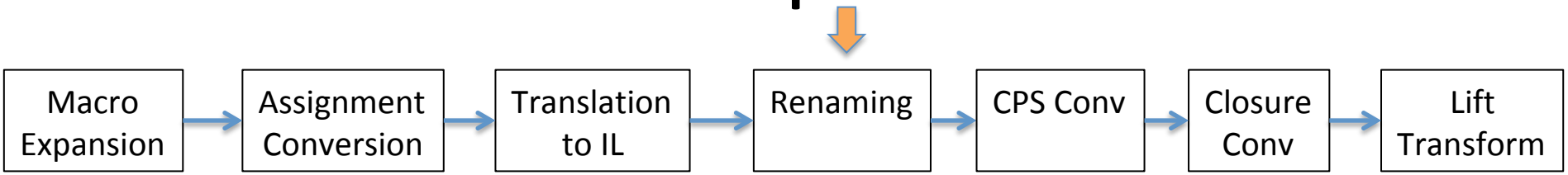

Compiler



Conversion to simple intermediate language without recursive forms

```
((prim-car print) "Hello World!")
```

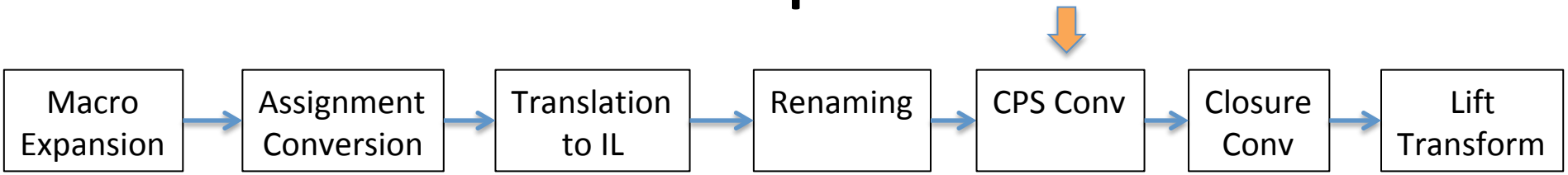

Compiler



To ensure uniqueness of variable names

```
((prim-car print) "Hello World!")
```

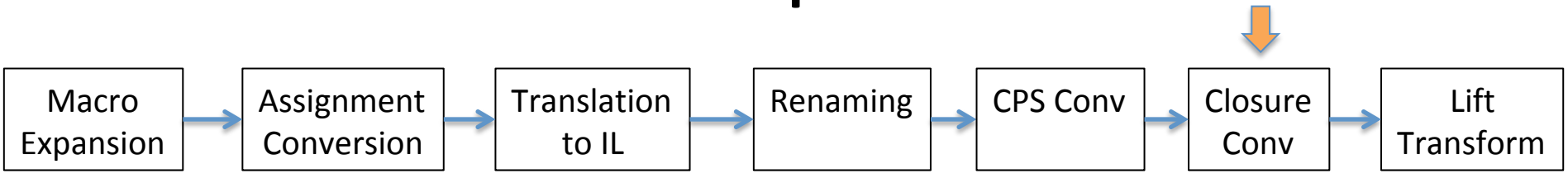

Compiler



Conversion of code to continuation passing style

```
(lambda (#:g00008073)
  (save-continuation #:g00008073)
  (let ((#:g00008074 (prim-car print)))
    (let ((#:g00008075 (lambda (#:g00008076)
                         (#:g00008073 #:g00008076))
                       (#:g00008074 "Hello World!" #:g00008075))))))
```

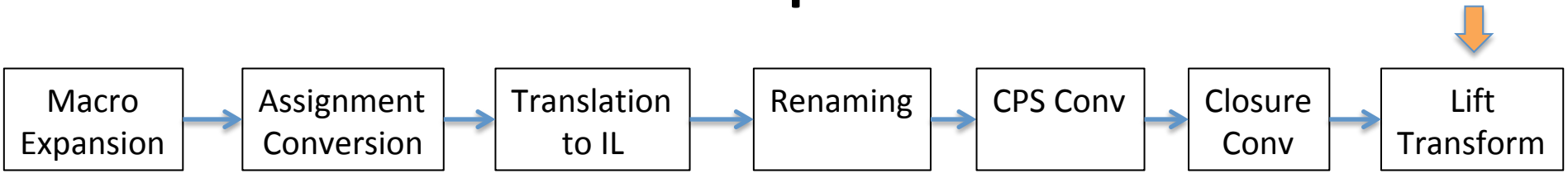

Compiler



Transformation of all functions to closures

```
(lambda (#:g00008077 #:g00008073)
  (save-continuation #:g00008073)
  (let2 ((print (nth1 1 #:g00008077)))
    (let ((#:g00008074 (prim-car print)))
      (let2 ((#:g00008081 (lambda (#:g00008078 #:g00008076)
                            (let2 ((#:g00008073 (nth1 1 #:g00008078)))
                              (let2 ((#:g00008079 #:g00008073)
                                     (#:g00008080 (extract-native-fn #:g00008079)))
                                (#:g00008080 #:g00008079 #:g00008076)))))
             (#:g00008075 (create-fn-closure 1 #:g00008081 #:g00008073)))
        (let2 ((#:g00008082 #:g00008074)
               (#:g00008083 (extract-native-fn #:g00008082)))
          (#:g00008083 #:g00008082 "Hello World!" #:g00008075))))))
```


Compiler

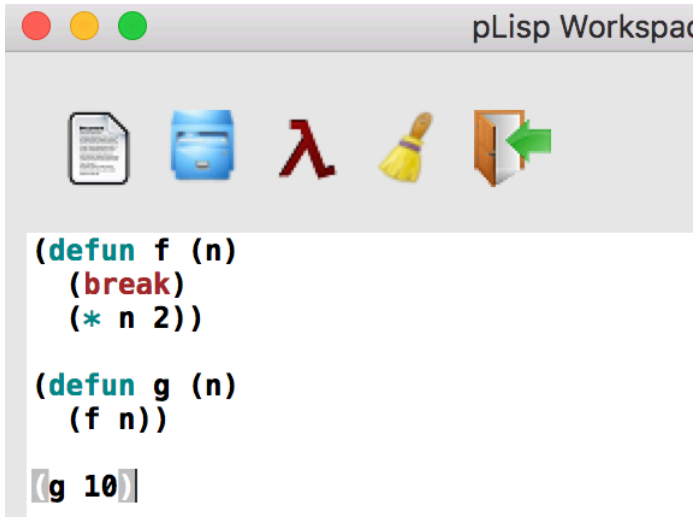


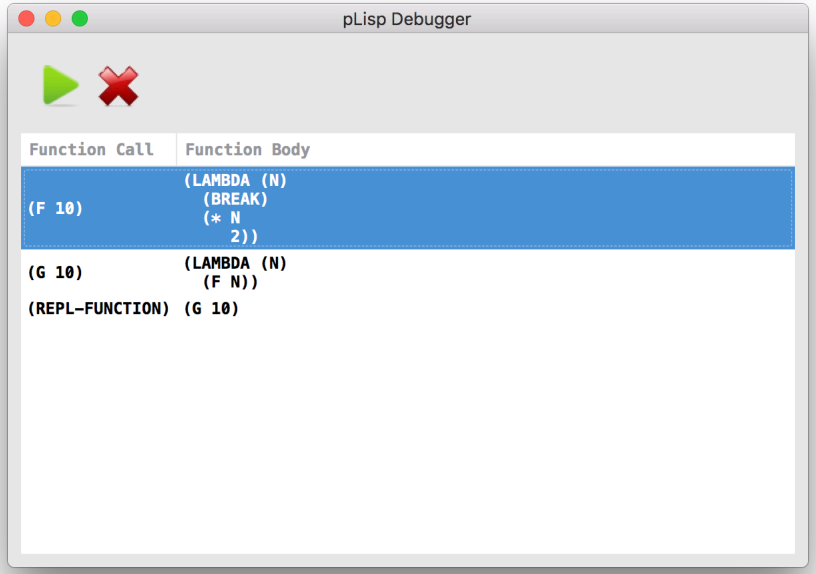
Eliminate function nesting and lifting all functions to the top level

```
(#:g00008084 (lambda (#:g00008077 #:g00008073)
               (save-continuation #:g00008073)
               (let2 ((print (nth1 1 #:g00008077)))
                 (let ((#:g00008074 (prim-car print)))
                   (let2 ((#:g00008081 #:g00008085)
                          (#:g00008075 (create-fn-closure 1 #:g00008081 #:g00008073)))
                     (let2 ((#:g00008082 #:g00008074)
                            (#:g00008083 (extract-native-fn #:g00008082)))
                       (#:g00008083 #:g00008082 "Hello World!" #:g00008075)))))))
(#:g00008085 (lambda (#:g00008078 #:g00008076)
               (let2 ((#:g00008073 (nth1 1 #:g00008078)))
                 (let2 ((#:g00008079 #:g00008073)
                        (#:g00008080 (extract-native-fn #:g00008079)))
                   (#:g00008080 #:g00008079 #:g00008076)))))
```


Debugger

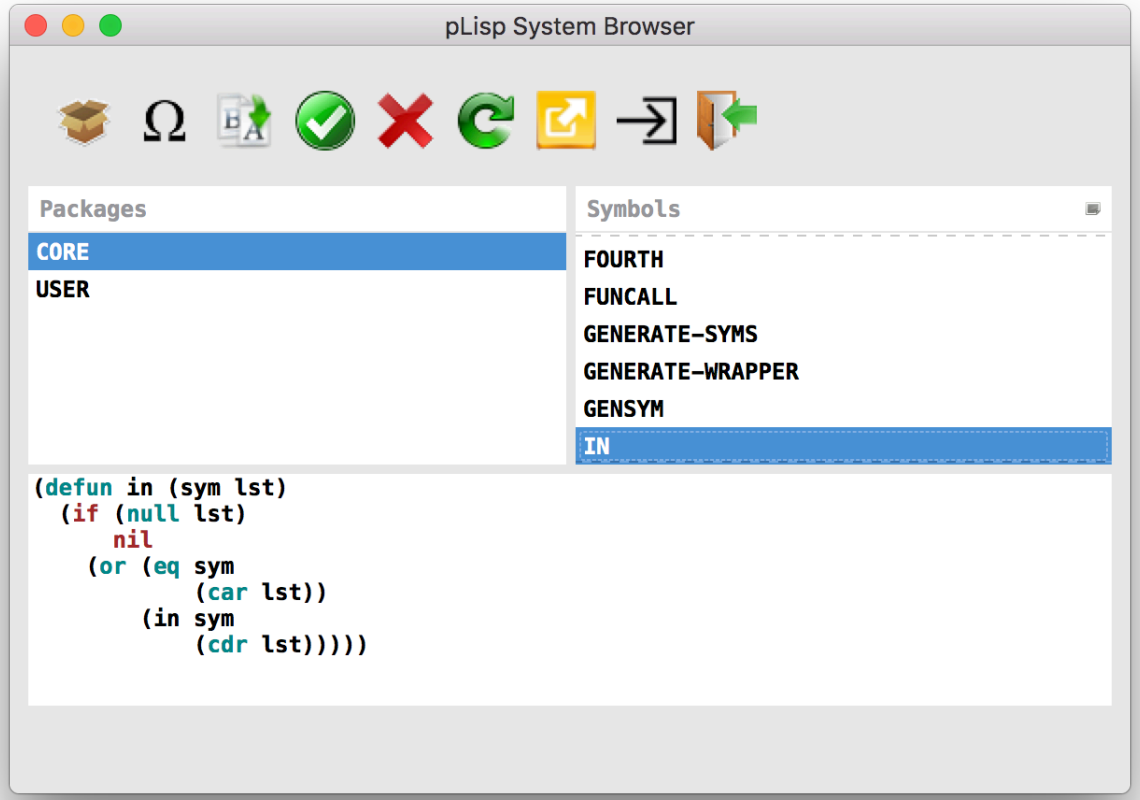
- Since we use CPS, all functions invoked for an expression evaluation are extant
- Debug stack is also a pLisp object
- Debug stack filters out the internal continuation functions generated by compiler
- At present, only break/resume/ inspection of function arguments supported
  - Continuing/restarting computation with user-supplied values, access to local variables are being considered for future work





pLisp Core Library

- Written in pLisp itself
- Wrappers for primitives so that they can be used as first-class functions (for use in constructs like map)



- Arithmetic operators
- Logical operators
- List operations
- String operations
- Array operations
- FFI
- Iteration/Looping

Serialization

- Image serialization in JSON format
- In addition to persisting objects
  - GUI elements
  - Shared libraries
  - Open pLisp source files
- Image size ~ 5 MB (uncompressed)
- Serialized objects via dummy pointers
  - References to a linear ‘heap’ in the JSON structure
  - Effectively equivalent to `OBJECT_PTR` in the live system
- Serialization of integers and floats
  - Image stores closure/macro code as C source
  - Since raw addresses do not carry over across programming sessions, C code cannot refer to the objects directly
  - Two options
    - Allocate space for ints/floats in our JSON heap
    - ✓ Generate integer/float constants by special calls (`convert_int_to_object()`,…)

Foreign Function Interface

- `load-library` and `call-foreign-function`
- Argument types supported
  - integers
  - floats
  - characters
  - pointers of above three types
- Return types: void, integers, floats, char pointers

```
gcc -c -fPIC test_so.c -o t
gcc -shared -W1,-soname,lib
```


Future Work

- Enhancements to the debugger
  - Continuing/restarting computation with user-supplied values, access to local variables
- Object Inspector
- Improve portability
  - Same image should be usable across different platforms
- Traceability between code version and image version

Demo

Q&A

Thank you!

Rajesh Jayaprakash
rajesh.jayaprakash@tcs.com