Extension Languages

Extension languages are often very high level **scripting languages** that are used to “glue” together larger software components, such as windowing software, network software, etc. The idea is to have an interpreter than can be extended by linking additional compiled libraries into the interpreter, thereby extending the virtual machine. You then define commands in the scripting language that can take advantage of the extended virtual machine.

Each scripting language has its own syntax and semantics, but has the ability to allow the programmer to “extend” the language by adding new commands that map onto compiled C or C++ code that reside in some library.

The extension language programs are all executed by running an interpreter for the language. For efficiency, some interpreters have built in JIT (Just in Time) compilers that compile scripts into bytecodes on-the-fly. For example, this is how the Tcl interpreter works (for Tcl 8.4)

The idea of being able to extend the syntax of a language is not new. The macro preprocessors provide a limited way to extend the syntax of a language. In Scheme and Lisp, the “define” operation allows you to define new operations that extend the basic set of operations in the interpreter.

However, extension languages like Tcl are unique in that the interpreter can be imbedded into another program. You can write a C or C++ program that is the controlling program for an application, and you can embed a Tcl interpreter into that program. In most Scheme/Lisp systems, the interpreter has to “be in charge” and you cannot embed the interpreter into another program.

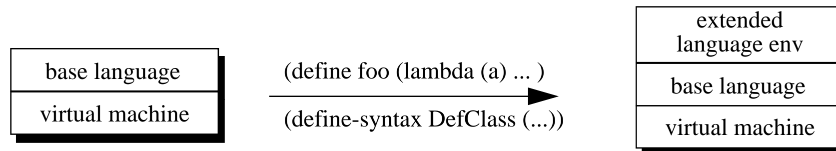
The Free Software Foundation has a project called GNU **Guile** that is based on the idea of an embeddable Scheme interpreter. The goal of Guile is to have a single virtual machine that can execute Scheme, Tcl, Java, or Ctax, which is a C-like syntax for writing Scheme programs. You can learn about Guile from www.gnu.org/software/guile.

Recently, people have leveraged the ubiquity of Java virtual machines to define new languages that compile down to Java bytecodes. This allows the language designer the freedom to focus on the language syntax and semantic model without having to worry about the execution environment. The language designer’s life is simplified by relying on the JVM as the only target “machine” for the language so she can concentrate on the language front-end. For example, check out Kawa Scheme, which is a scheme system that compiles to Java bytecodes (http://www.gnu.org/software/kawa). Another interesting idea is to extend Java syntax itself using Java’s reflective programming capabilities. If you are interested, check out OpenJava (http://www.csg.is.titech.ac.jp/openjava)

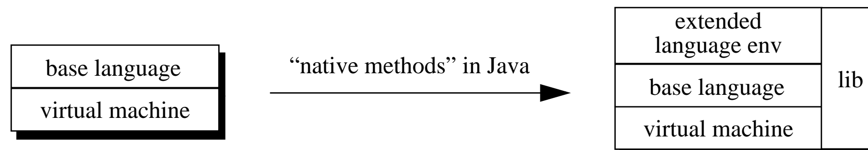
Extension Languages

There are various ways to build extensions, depending on how the extension language is designed. The most simple approach is to allow **definitional extension**. A more powerful approach is to allow dynamic linking with external functions in some C/C++ library, and define names in the extension language that map onto those “native” functions. The third approach is to extend the virtual machine itself, by adding some new capabilities to the core set of primitives that were not previously available. For example, adding threads to a VM to allow concurrent programs

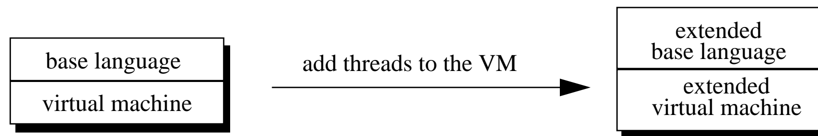
.



(a) extending the language environment using the base language



(b) extending the language environmentby linking with “foreign” functions written in C/C++



(c) extending the base language by extending the virtual machine

Domain Specific Embedded Languages

Scheme, LISP, ML, and Haskell allow definitional extension by extending core primitives. The interpreter semantics are fixed, and the programmer can define new procedures using the builtin primitives and other command/ procedures defined in the global environment.

Perl, Java, Tcl, and Python are languages that allow linking (either statically or dynamically) with external libraries, so that you can link in external C/C++ code. You then define new commands in the scripting language environment.

Extending the VM requires that the VM be constructed in such a way that it allows extension. This approach is still a topic for programming language research.

This approach is generally called ***Domain Specific Embedded Languages (DSELs)***, since the goal is to extend a core set of primitives of an existing language with new syntax ()and maybe new semantics) for application to a specific domain (e.g., bioinformatics), effectively embedding a DSL into an existing language framework.

Syntactic extension can be accomplished in several ways:

- Use builtin **definitional** or **macro syntax extension** facility to introduce new syntactic primitives, e.g., hygienic macros using define-syntax in Scheme or definitional extension in Haskell and ML.
- Utilize a language's builtin **polymorphic definition mechanisms** to define new "named objects" that are then used within the framework of the encompassing (general purpose) language as a kind of embedded sub-language. For example, using C++ templates and operator overloading or Java 1.5.
- Extend the interpreter for an existing language by embedding the core interpreter into an **extended interpreter** for the new DSL. For example, using GNU Guile.
- Build a **common language run-time** framework that can be used as the execution target for a variety of front-end languages that are each "compiled" to this CLR. For example, Microsoft's CLR.
- Implement a **source-to-source translator** that translates an extended syntax into the source of a target language (for example, using OpenJava).

Syntactic Extension in Scheme

Scheme (i.e., standard Scheme as specified in R5RS - Revised(5) Report on the Algorithmic Language Scheme) has a powerful macro facility that allows for a variety of syntactic extensions. For example, the let and let* expressions in Scheme are syntactic macros built up from the more primitive lambda form:

```
(define-syntax let
  (syntax-rules ()
    ((_ ((x v) ...) e1 e2 ...)
     ((lambda (x ...) e1 e2 ...) v ...))))
```

The let* expression is defined similarly. A “named let” is a bit more complicated.

Other simple examples include:

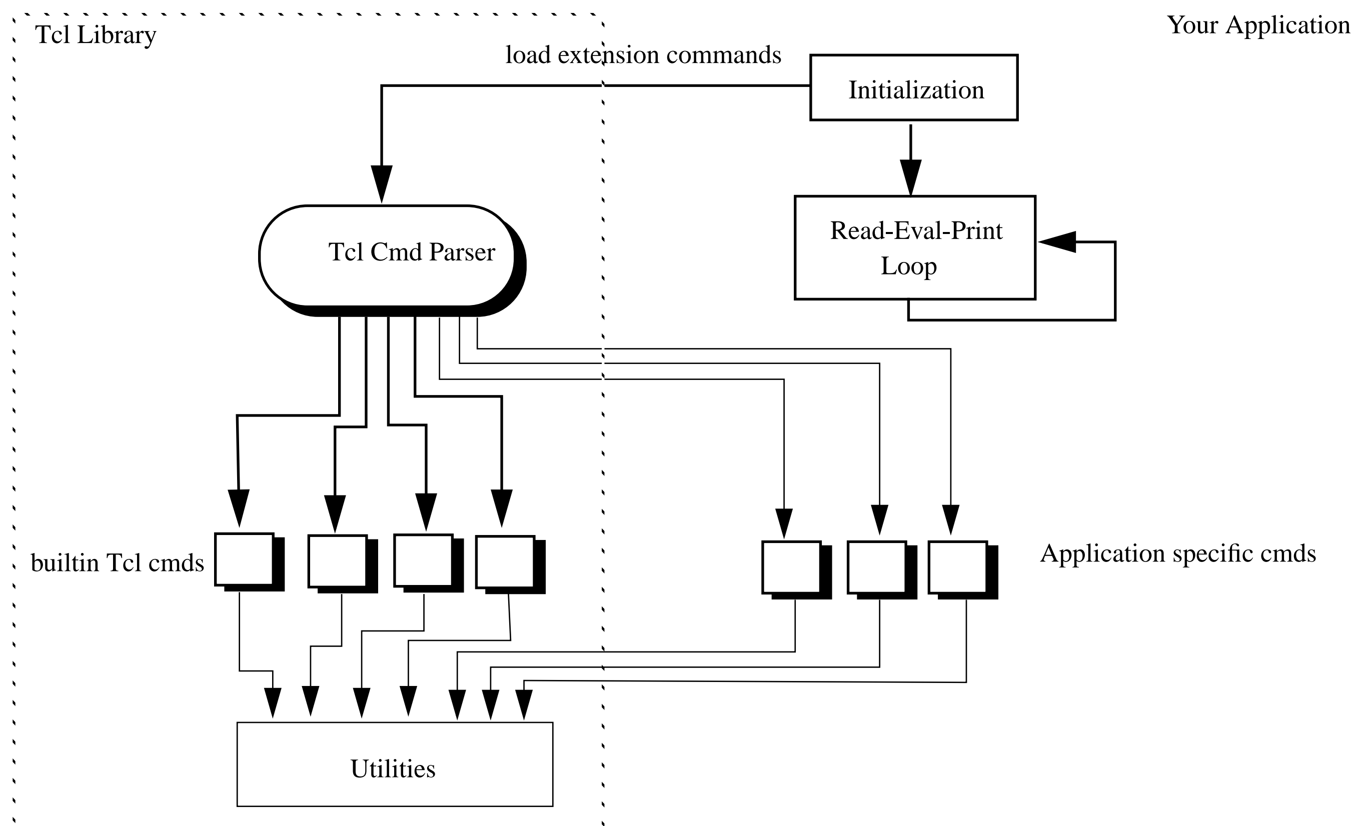
```
(define-syntax and
  (syntax-rules ()
    ((_) #t)              ;; (and)
    ((_ e) e)             ;; (and e)
    ((_ e1 e2 e3 ...)     ;; (and e1 e2 e3 ...)
     (if e1 (and e2 e3 ...) #f))))

(define-syntax or
  (syntax-rules ()
    ((_) #f)               ;; (or)
    ((_ e) e)              ;; (or e)
    ((_ e1 e2 e3 ...)      ;; (or e1 e2 e3 ...)
      (let ((t e1))
        (if t t (or e2 e3 ...))))))
```

A question we will analyze and investigate is far can we go with this type of syntactic extension based on the core semantics of the Scheme interpreter.

Example: The Tcl Interpreter

The Tcl interpreter is written in C and is part of a C library that you can link into your C/C++ program. The “wish” shell is just a simple command line interface to a Tcl interpreter that has been extended to include the X-Windows library on Unix, the Microsoft Foundation Class (MFC) library on Windows, and the MacOS X window system. The nice thing is, you don’t have to be an expert at programming with X-Windows or MFC in order to write useful GUI applications. You just have to learn some simple Tcl and Tk commands, write your application, and it will then run on Unix, Windows, and MacOS!

Here is a diagram of how the Tcl interpreter is embedded in an application and extended with new commands:



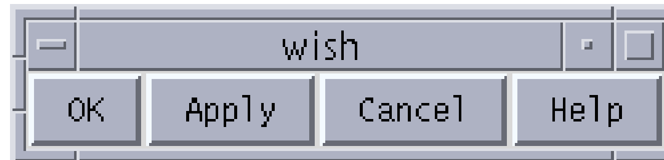
Example Using TCL/Tk

TCL stands for *Tool Command Language*. Tcl was developed by Dr. John Ousterhout when he was a Computer Science professor at UC Berkeley. He has now gone on to fame and fortune as a Research Fellow at Sun Microsystems, where he leads the design and development of Tcl and a powerful windowing extension called Tk, both of which are freely available. You can download Tcl/Tk from http://www.scriptics.com

Tk, which stands for Windowing ToolKit, works with the common windowing systems on Unix, Windows and MacOS. You can quickly build GUI applications using the Tcl/Tk combination. Here is an example of some Tk commands typed in at the Tcl interpreter prompt. The extended Tcl interpreter is called "wish" for Windowing Shell.

```
wish% button .ok -text OK -command ok
wish% button .apply -text Apply -command apply
wish% button .cancel -text Cancel -command cancel
wish% button .help -text Help -command help
wish% pack .ok .apply .cancel .help -side left
```

The window that results from these simple commands is:



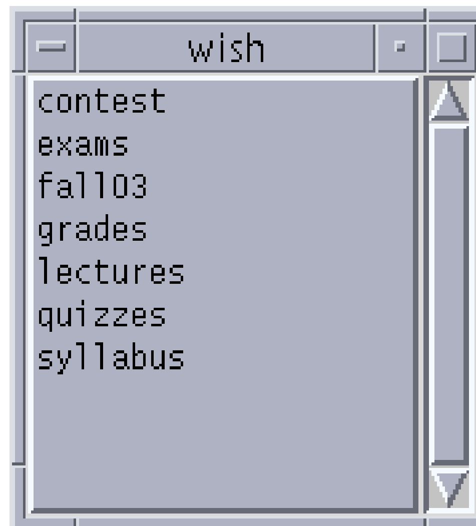
The commands "button" and "pack" are not Tcl commands, but syntact extensions to the base Tcl language that allow a programmer to invoke windowing system semantics using a very high-level syntax. This new syntax corresponds to procedures written in C that create widgets for the window system. The commands are interpreted by the Tcl interpreter, which doesn't recognize them as builtin commands as part of the normal Tcl language, so the Tcl interpreter checks to see if the commands are defined as part of an extension to the virtual machine. The arguments are then passed to the C procedure that corresponds to the commands. The -command flag to the button procedure defines another procedure that will be called when a button is pressed. In windowing system terminology, these procedures are "call back" procedures, that will be called when a button is pressed.

More Tk Commands

You can easily define a window that allows you to scroll through a list of files in a directory by combining Tcl procedures with Tk windowing commands. For example:

```
wish% listbox .files -relief raised -borderwidth 2 -yscrollcommand ".scroll set"
wish% pack .files -side left
wish% scrollbar .scroll -command ".files yview"
wish% pack .scroll -side right -fill y
wish% foreach i [lsort [glob *]] {
>        .files insert end $i
>     }
```

The result is a window for a listbox with a scrollbar that lists all the files in the current directory. In this case, the directory containing files and directories from a previous course::



To learn more about Tcl/Tk programming, I recommend the following book: *Practical Programming in Tcl and Tk, 4th Edition*, by Brent Welch, et. al.

Native Code Extension in Java

Java provides the Java Native Interface as a way to extend the functionality of Java by adding extension into the Java virtual machine run-time. The java.lang.Math class and the java.lang.Thread class and the java.lang.Socket class are examples of language features built up by linking to standard C libraries that implement these features. They have become part of the core language of Java, but Java can be extended by adding in class that are implemented in terms of an extended virtual machine. In a manner of speaking, Java has a core set of primitives and is extended through many class libraries that are built up out of this set of core primitives.

```
public class Thread implements Runnable {
     ...
     public synchronized native void start();
     public static native Thread currentThread();
     public static native void yield();
     public static native void sleep(long millis) throws InterruptedException;
     public static int enumerate(Thread tarray[])

     public static boolean interrupted() { ... }
     public boolean isInterrupted() { ... }
     public final native boolean isAlive();
     public String toString() {
     public void interrupt() { ... }
     public void interrupt() { ... }
     public final void stop() { ... }
     public final void suspend() { ... }
     public final void resume() { ... }
     public final void setPriority(int newPriority) {
     public final int getPriority() {
     public final void setName(String name) { ... }
     public final String getName() { return String.valueOf(name); }
     public native int countStackFrames();
     public final synchronized void join() throws InterruptedException {...}
     public void destroy() { throw new NoSuchMethodError(); }
```