PL Features: Reflection
PL Categories: Lisp-Family PLs
Introduction to Scheme

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Wednesday, March 29, 2017

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

© 2017 Glenn G. Chappell
Recall: a **compiler** takes code in one PL (the **source** PL) and translates it into code in another PL (the **target** PL).

An **interpreter** takes code in its source PL and executes it.
Compilation and interpretation are not mutually exclusive. Many modern interpreters begin by compiling to an intermediate representation (IR) — perhaps a byte code — which is then interpreted directly.
The AST produced by the parser will need to be processed, either by interpreting it directly, or generating another IR from it. How is this done?

An AST is a **rooted tree**. Code that deals with a rooted tree usually proceeds as follows.

- Handle the root node.
- Make a function call (often a recursive call) on each subtree of the root.

```
(a + 2) * -b
```

![AST Diagram]

29 Mar 2017
Suppose we wish to write a function that evaluates an AST representing a numeric expression, like the pictured tree. Our function will take an AST and return the numeric value of the expression.

It could work something like this:

- If the root node represents a numeric literal:
  - Convert the literal to a number and return it.
- Else if the root node represents a numeric variable:
  - Get the variable’s current value and return it.
- Else if the root node represents a binary operator:
  - Get the value of the left subtree (recursive call).
  - Get the value of the right subtree (recursive call).
  - Apply the appropriate operation and return the result.
- Else if the root node represents a unary operator:
  - Get the value of the subtree (recursive call).
  - Apply the appropriate operation and return the result.
While an interpreter is executing a program, there will need to be some representation of program state: values of variables, the call stack, etc.

In a PL with static typing and scope, the compiler/linker can determine the types and scopes of all variables and the types of all unnamed values. These can be laid out in memory (for local values, in a stack frame). Thus, at runtime, a reference to a value will simply be a reference to a particular memory location.

In a dynamic PL, it is common to place variables in an associative structure with the variable name as key. Usually a hash table is used, with a separate hash table for each scope.
There will need to be a **runtime system** (often simply **runtime**): additional code that programs will need to use at runtime. This might include:

- Program initialization and shutdown.
- I/O.
- Memory management.
- Interfaces to operating system functionality (e.g., files, threads, interprocess communication).
- Implementations of PL commands that perform complex operations (e.g., advanced floating-point computations, operations involving multiple data items like sorting or matrix operations).
PL Features: Reflection
What It Is

**Reflection** in a computer program refers to the ability of the program to deal with its own code at runtime: examining the code, looking at its properties, and modifying it.

In practice, reflection is largely a property of a programming language, along with the runtime environment that supports its execution. If a PL supports examining and modifying code written in that PL, and then executing the modified code as part of the execution of the same program, then that PL will generally allow for reflection in programs written in the PL.

Thus, an important property of a PL is whether, and how well, it supports reflection.
PL Features: Reflection
Support in Various PLs

PLs like C, C++, and Java offer no support for reflection.

Dynamic PLs like Lua usually have some reflection support. Objects are typically hash tables that can be modified at runtime.

There is a long history of reflection in functional programming, so it is curious that Haskell does not support reflection. I have seen discussions about this, but I do not understand the issues well.

Concatenative PLs typically offer decent reflection support. For example, in Forth, `see` shows a word’s source.

The gold standard for reflection support is the Lisp family of PLs. Lisp programs use reflection as a matter of course, modifying their own code via transformations called `macros`.

29 Mar 2017
We will shortly begin a study of a Lisp-family PL called **Scheme**. As a Lisp-family PL, Scheme offers excellent support for reflection.

**TO DO**
- Look at some examples of reflection in Scheme.

Done. See `reflect1.scm`. 
In 1958, MIT professor John McCarthy published a mathematical formalism for describing computation. This formalism was written in various ways; the most common notation was the **Symbolic Expression**, or **S-expression**. An S-expression is either an **atom** (basically a word), a **pair** (two S-expressions separated by a dot and enclosed in parentheses), or **nil** (an empty pair of parentheses).

```
(THIS . (IS . (AN . (((S . (EXPRESSION . ())) . ()) . ())))))
```

A shorter form uses a parenthesized list of space-separated items. Something like “(A . (B . (C . ())))” is written as “(A B C)”.

```
(THIS IS AN (S EXPRESSION))
```
Accounts vary concerning what happened next. Apparently, Dartmouth student Steve Russell, having read McCarthy’s paper, observed that one of the operations of the formalism—“eval”—could be implemented as a computer program. The result would be an interpreter for a programming language, and S-expressions would be programs.

In 1958, Russell wrote such an implementation in machine language on an IBM 704. The resulting PL became known as Lisp, for LISt Processor.

Lisp and associated PLs are noted for their excellent support for reflection. Code and data are stored in the same structures (binary trees holding S-expressions). Construction and modification of code at runtime is common.
Lisp caught on rapidly in the Artificial Intelligence research community. By the 1970s several dialects were in use.

In the late 1960s, a Lisp dialect called Logo was released. This was aimed at teaching programming concepts.

The mid-1970s saw the development of EMACS (originally Editor MACroS) a Lisp-scriptable text editor. A descendant continues to be actively developed by the GNU project; it is widely used.

In the 1980s a computer called a lisp machine, aimed at running Lisp, was sold. Lisp machines are no longer made.

A number of important PL concepts had their first major implementations in Lisp-family PLs: recursion, tree structures, closures, dynamic typing, higher-order functions (including encapsulated loops like map, filter, zip, fold/reduce), garbage collection, and REPLs.

Interest in Lisp died down in the 1990s. But Lisp appears to be enjoying a resurgence.

Of particular interest is Clojure (pronounced like “closure”) a Lisp dialect for the Java Virtual Machine (JVM). Originally written by Rich Hickey in 2007, Clojure continues to be actively developed.
Lisp-family PLs typically have the following characteristics.

- Simple syntax based on the S-expression. A program is a list of lists. The first item of a list is a function; the rest are its arguments.
  - For example, the C/C++/Java expression \((a + 2) * -b\) would be written as follows in a typical Lisp-family PL:
    \[ (* (+ a 2) (- b)) \]
  - Where have we seen this before?
  - Tweak the notation a bit: replace parentheses with braces, separate list items by commas, and place atoms in double quotes. Result:
    \{"*", "{"+", "a", "2"}, {"-", "b"} \}
  - Where have we seen this before?
  - This was our first stab at an AST representation in Lua.
  - Thus, **Lisp source code is a direct representation of its own AST!**
- Source-code syntax and storage format is that of the PLs primary data structures. Types can be checked, and code can be modified & executed at runtime. Support for reflection is excellent.
- Typical programming styles involve **macros**: transformations applied to code at runtime.
Typical characteristics of Lisp-family PLs (cont’d):

- Typing is dynamic, implicit, and structural (duck typing).
- There is very good support for functional programming: first-class functions, higher-order functions, etc. But mutable data *is* allowed.
- The PL is extensible.
- Execution can be either interactive (REPL) or via previously compiled code.
- Accomplished Lisp programmers tend to be insufferably fond of Lisp. (“We can already do that in Lisp. <smirk>”) But—perhaps they’re onto something.
As with many PLs, the early history of Lisp was one of ever-increasing complexity. As a result, the Common Lisp standard is huge, including exceptions, an object system, etc.

Perhaps as a reaction to this, a Lisp-family PL called Scheme was created at the MIT AI Lab around 1970, by Guy Steele and Gerald Sussman.

In contrast to Common Lisp, Scheme follows a minimalist design philosophy, with a small, simple core and versatile tools for extending the PL.
Scheme follows somewhat different conventions from traditional Lisp for the evaluation of functions. Thus, while some say Scheme is a dialect of Lisp, others emphatically deny this. But Scheme clearly belongs in the Lisp family of PLs.

Scheme has been standardized in a series of standards documents. The most recent (R7RS) was released in 2013.

A version of Scheme called PLT Scheme (named for the Rice University Programming Languages Team) was first released in 1994. This was renamed as Racket in 2010. Distributed with Racket is a simple IDE called DrRacket, which runs on all major platforms. This is the Scheme implementation we will be using.
Introduction to Scheme
Characteristics — Introduction

Scheme is a Lisp-family PL with a minimalist design philosophy.

Scheme code consists of parenthesized lists, which may contain atoms or other lists. List items are separated by space; blanks and newlines between list items are treated the same.

```
(define (hello-world)
  (begin
    (display "Hello, world!")
    (newline)
  )
)
```

When a list is evaluated, the first item should be a procedure (think “function”); the remaining items are its arguments.
The type system of Scheme is similar to that of Lua.

- Typing is dynamic.
- Typing is implicit. Type annotations are generally not used.
- Type checking is structural. Duck typing is used.
- There is a high level of type safety: operations on invalid types are not allowed, and implicit type conversions are rare.
- There is a fixed set of types.

Lua’s fixed set of types includes only 8 types, while Scheme has 36. We look at some of these next.
Two heavily used types are **pair** and **null**, which are mostly used to construct lists.

Values of all other types are **atoms**. Here are a few of these:

- **Booleans.** Values are `#t` (true) and `#f` (false).
- **Strings.** Enclosed in double quotes: "This is a string."
- **Characters.** For example, here is the 'a' character: `\a`
- **Symbols.** A **symbol** is an identifier: `abc x a-long-symbol`
- **Number types.** There are seven of these, including arbitrarily large integers (like Haskell’s `Integer`), floating-point numbers, exact rational numbers, and complex numbers.
- **Procedure types.** A **procedure** is what we would call a first-class function. A procedure may be bound to a name (a symbol), or it may be unnamed. **There are actually six procedure types, but we will not need to distinguish between these.**
Scheme has no special syntax for flow of control. Instead, flow-of-control constructs are procedures. Here is some Lua code and more or less equivalent Scheme code.

```lua
if x == 3 then         -- Lua
    io.write("three")
else
    io.write("other")
end
```

```scheme
(if (= x 3) ; Scheme – "if" is a procedure
    (display "three")
    (display "other")
)
```
As with Lua, Scheme local variables are lexically scoped. Scheme globals have dynamic scope.

Scheme has very good support for functional programming. It is not a pure functional PL; it does allow for mutable data.

Like all Lisp-family PLs, the syntax and storage format of code is the same as that of the language’s primary data structure. Thus, it is natural to manipulate code at runtime. Such code can be executed as part of the same runtime. Thus, Scheme has excellent support for reflection.
The standard filename suffix for Scheme source files is `.scm`.

Scheme allows for interactive execution or compiled executables. We will not be doing the latter.

We will execute Scheme using an IDE called **DrRacket**.

- The upper part of the DrRacket window is a source-code editor, with the usual open-save interface. The first line of the code in this window should always be as follows:

```
#lang scheme
```

- The lower part of the Window is a REPL. Type in Scheme code to execute.
- Pressing the “Run” button executes all code in the upper window. Thereafter, symbols defined in that code may be used in the REPL.
As usual, a Fibonacci-number printing program is available.

See fibo.scm.